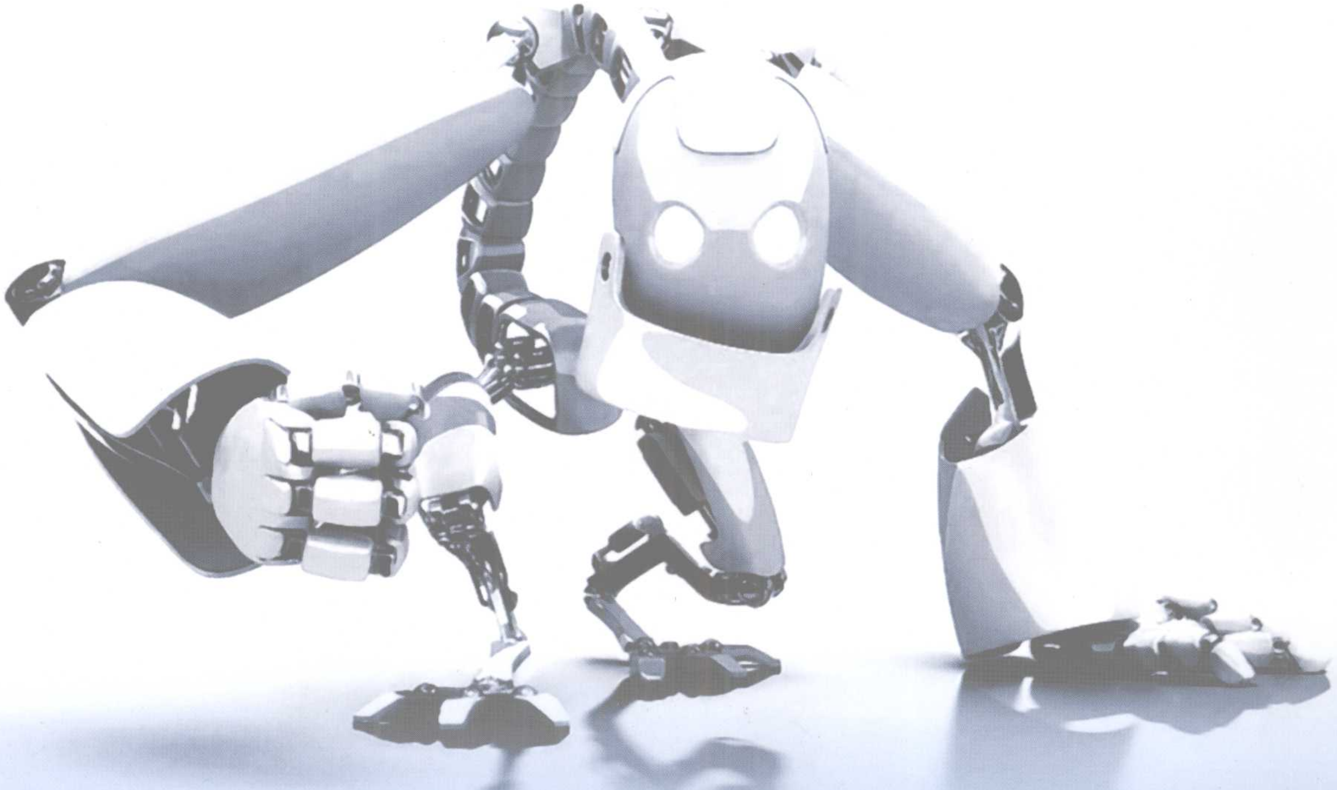
HZ BOOKS

Windows Embedded CE 6.0开发指南及应用构建流程

嵌入式技术丛书

Windows Embedded CE 6.0 程序设计实战

姜波　编著



附赠光盘

机械工业出版社
China Machine Press

Windows Embedded CE 6.0 程序设计实战

本书特色

- 内容详实，涵盖了Windows Embedded CE 6.0程序设计的基础知识和程序设计技巧。
- 介绍了Visual Studio 2005集成开发环境的使用方法，及使用Visual C# 2005和Visual Basic 2005构建Windows Embedded CE 6.0下游戏程序的过程。
- 介绍了使用Platform Builder for CE 6.0进行Windows Embedded CE 6.0内核定制的过程，且所有程序代码都运行在这一自定义的模拟器上。

配套光盘 附赠光盘

包含了Windows Embedded CE 6.0版“连连看”、“俄罗斯方块”等多种游戏程序实例。同时，所有的程序实例在书中均有详细的代码解释，使读者能够触类旁通，利于启发和培养读者的创造力。

关于作者

姜波　微软嵌入式领域最有价值专家（Microsoft Embedded MVP），多年从事Windows CE及Pocket PC程序设计工作，所开发的基于Windows CE的微小型机器人控制软件已经应用于教学、微小型机器人控制等多个领域。

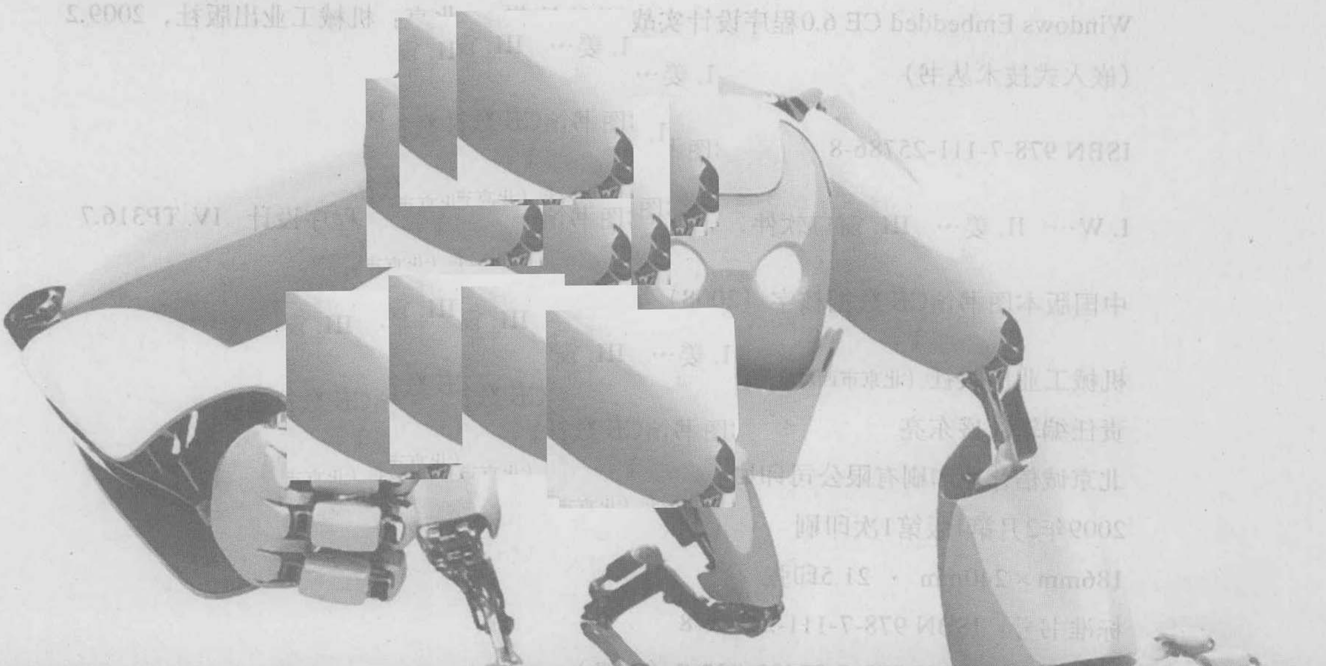
投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com
华章网站 http://www.hzbook.com
网上购书：www.china-pub.com
封面设计·王建敏

上架指导：计算机/嵌入式
ISBN 978-7-111-25786-8
9 787111 257868
定价：55.00 元（附光盘）

嵌入式技术丛书

Windows Embedded CE 6.0 程序设计实战

姜 波 编著



机械工业出版社
China Machine Press

本书系统介绍了嵌入式系统程序设计方法。全书共18章，分别介绍以下内容：Windows Embedded CE 6.0程序设计的基础知识（第1～7章）；Windows Embedded CE 6.0程序设计的实例（第8～14章）；基于Windows Embedded CE 6.0的高级程序设计（第15～18章）。

本书内容体系完整，案例详实，叙述风格平实、通俗易懂。尤其适合嵌入式系统研究方向的高年级本科生、研究生及科研技术人员参考。

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

图书在版编目（CIP）数据

Windows Embedded CE 6.0程序设计实战/姜波编著. —北京：机械工业出版社，2009.2
（嵌入式技术丛书）

ISBN 978-7-111-25786-8

I. W…　II. 姜…　III. 窗口软件，Windows Embedded CE 6.0－程序设计　IV. TP316.7

中国版本图书馆CIP数据核字（2008）第201311号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：盛东亮
北京诚信伟业印刷有限公司印刷
2009年2月第1版第1次印刷
186mm×240mm · 21.5印张
标准书号：ISBN 978-7-111-25786-8
　　　　　ISBN 978-7-89482-920-7（光盘）
定价：55.00元（附光盘）

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：（010）68326294

前言

20世纪，以个人电脑为代表的信息技术的迅速发展，对世界的政治、经济、军事和科学技术产生了深远的影响，而由此产生的信息产业则成为世界各国经济增长的动力之一。同时，伴随着生产力的不断进步和新兴产业的不断兴起，各种具有高技术含量、高生产效率、低运营成本的技术被越来越多地应用于各个行业，而嵌入式系统则是其中的典型代表。

所谓嵌入式系统，主要是指以应用为中心，以计算机技术为基础，软硬件可以根据需要进行裁剪，适用于对系统的成本、体积、性能、功耗等有严格要求的专业计算机系统。它一般由嵌入式微处理器、外围硬件设备、嵌入式操作系统及应用程序等四个部分组成。从日常生活中的电视机顶盒、智能手机，到航空航天飞行器、汽车电子、网络通信、数据传输，随处都有嵌入式系统的身影。在刚刚成功举办的2008年北京奥运会中，奥运场馆的人员安检、通信指挥系统的部署、运动员比赛场地的清理等各个领域都有大量的嵌入式设备工作其中。可以说，我们的生活正在不知不觉地被嵌入式系统改变着。

那么基于嵌入式系统的产品和传统产品相比，具有哪些特点呢？首先，基于嵌入式系统的产品一般都是由软件和硬件所组成，比如现在流行的GPS导航仪等便携式设备，既需要硬件显示屏幕，又需要有导航软件的支持；其次，嵌入式产品的开发和设计往往需要专业的开发工具，比如我们在本书中重点介绍的Visual Studio 2005和Platform Builder for CE 6.0等；再次，从产品的应用领域上看，原来那些使用PC机进行操作的地方几乎都可以用相应的嵌入式产品来替代，比如石油勘探领域的扭矩测量、电力领域的线路巡检、安检领域的身份识别等。同时，嵌入式产品的大规模、多层次运用还极大地提高了整个系统的实时性。比如，我们可以在不同位置安放大量的小型嵌入式产品，来实时地获取所需数据，从而取代过去那种分散数据采集、再由中央计算机集中分析的信息处理方式。最后，从用户体验上看，丰富多彩的嵌入式产品给我们的工作和生活都带来了极大的便利。以目前流行的基于Windows CE系统的智能手机为例，其CPU频率已经达到了600 MHz，内置存储器达到8 GB的存储空间，同时具备网络信息浏览、通话、拍照、GPS导航、Office软件办公、多媒体影音播放等多种功能，这些都极大丰富了我们的生活，同时也提高了我们的工作效率。

在众多的嵌入式产品中，被广泛采用的嵌入式操作系统主要有嵌入式Linux、Windows CE、Symbian、VxWorks以及Google最新推出的Android等。而本书重点介绍的Windows Embedded CE 6.0则是目前Windows CE产品家族中应用最为广泛的版本之一。Windows Embedded CE 6.0是一个具有抢先式多任务功能，并具有强大通信能力的嵌入式操作系统。也是微软公司专门为移动设备和消费类电子产品、嵌入式应用等非PC领域设计的全新操作系统。因此，根据应用环境的特点，Windows Embedded CE 6.0具有高度模块化、良好实时

性、强大通信能力、支持多种CPU的嵌入式操作系统。它主要具有以下特点：

- 良好的电源管理功能：很多嵌入式产品都是依靠电池来供电的，因此作为一款嵌入式操作系统，Windows Embedded CE 6.0具备了灵活的电源管理功能，包括睡眠/唤醒模式等。
- 良好的多任务管理功能：和Windows XP、Windows Vista类似，Windows Embedded CE 6.0同样具备良好的多任务管理功能。我们可以同时运行多个程序，并在这些程序之间自由切换。
- 人性化的图形用户界面：Windows获得成功的原因之一就在于其使用了友好的图形用户界面，Windows Embedded CE 6.0同样具备了此类功能，当我们启动它的时候，不但会看到类似于桌面PC的操作系统界面，系统的使用方式和文件命名方式也同桌面Windows基本一样。
- 丰富的通信功能：Windows Embedded CE 6.0的通信模块主要支持串行端口通信、红外通信、网络通信和蓝牙通信等。我们可以利用这些功能灵活地设计出符合实际需要的产品来。
- 优秀的多媒体扩展性：用户可以通过内置的MediaPlayer播放器播放音频和视频文件，同时还加入了对SD卡、CF卡等移动存储设备的支持。
- 强大的实时功能：Windows Embedded CE 6.0是一款实时性操作系统，因此可以被应用在对实时性要求较高的领域。

那么如何设计Windows Embedded CE 6.0的应用程序，Windows Embedded CE 6.0具有哪些特点，它和传统的桌面Windows相比又有什么样的不同呢？这就是我们在本书中要解决的问题。

本书全面而系统地介绍了Windows Embedded CE 6.0程序设计的基本知识及程序设计的方法。不仅在理论上详细介绍了Windows Embedded CE 6.0的特点，还通过多个程序实例的构建和代码讲解，介绍了使用Visual Studio 2005进行Windows Embedded CE 6.0程序设计和使用Platform Builder for CE 6.0进行Windows Embedded CE 6.0内核定制的基本步骤和方法。读者在了解基础理论的同时，可进一步提高实际动手能力。

本书共分为三大部分18章，具体内容安排如下：

第一部分介绍Windows Embedded CE 6.0程序设计的基础知识，包括第1章到第7章。主要介绍了什么是嵌入式系统，嵌入式系统的特点，Windows Embedded CE 6.0系统及其特点，Windows Embedded CE 6.0下的内存管理和进程、线程的特点，以及文件系统和注册表管理。

第二部分为Windows Embedded CE 6.0程序设计实例，包括第8章到第14章。读者可以在这一部分内容中学到如何使用Platform Builder for CE 6.0进行内核定制，如何在Windows Embedded CE 6.0设备中实现图形绘制和输入处理，如何使用Visual C# 2005设计“连连看”游戏程序，如何使用Visual Basic 2005设计俄罗斯方块游戏以及使用Embedded Visual C++设

计程序的过程。通过这一部分的学习，读者将会对Windows Embedded CE 6.0程序设计过程有一个较为深入的了解，同时能够熟练运用Visual Studio 2005和Platform Builder for CE 6.0。

第三部分为Windows Embedded CE 6.0高级程序设计部分，包括第15章到第18章。在这里我们为读者介绍了Windows Embedded CE 6.0的中断处理过程、驱动程序、流驱动程序和注册表项以及总线等，为读者进一步学习Windows Embedded CE 6.0程序设计打下更为深厚的基础。

我们在进行Windows Embedded CE 6.0相关软件的开发过程中，深切地体会到，只有不断地实践才会把所学到的知识很好地运用到实际工作中。一些在书本上看似简单的原理，也只有在应用中才会体会到其精髓所在。因此，建议读者在阅读本书的同时，最好按照书中的操作步骤，亲自把整个程序构建一次；同时还可以尝试修改程序中的某段代码，看看其效果和原来有何不同，或者把自己的想法和疑问发到我们的网站上，和大家进行交流……，我想，这些都会加深读者对Windows Embedded CE 6.0程序设计的理解，切实提高大家的动手能力和思考问题的能力。

一本好书的出版需要很多人的共同努力，本书在编写过程中得到了众多朋友的大力支持。在此要感谢北京汉邦高科数字技术有限公司的宁立君和北京六合伟业科技有限公司的贺健，他们为本书部分硬件相关的内容提供了大量技术资料，对技术难点给出解释；要感谢我的良师益友、北京航空航天大学软件学院副院长、嵌入式软件专业主任康一梅博士对本书的总体结构所提出的多项改进意见；还要感谢北京中海纪元数字技术发展有限公司天津分公司总经理魏冰、北京数天辰科技发展有限公司市场部经理赵志波、微软（北京）.Net俱乐部主席葛涵涛、北京航空航天大学软件学院网络工程师李建伟为本书内容所提供的大量实际项目经验和市场化运作建议。

在这里我要特别感谢我的父母和妻子，是他们在本书的编写过程中给予了我极大的鼓励，是他们的牵挂陪伴我度过了多少个日日夜夜，也是他们默默地支持坚定了我完成本书的信心，感谢他们为我所做的一切。

本书作者多年从事嵌入式平台研发、IT书籍编写等工作。作为第一作者编写有《Windows CE.Net程序设计》、《Visual C# 2005智能设备程序设计》、《Visual Basic 6.0/.Net游戏开发实例》等书籍。其中《Windows CE.Net程序设计》一书是中国电子学会嵌入式认证考试系列教材之一，同时也是中国电子学会嵌入式专家委员会的指定教材。

另外，参与本书编写的还有李淑云、杜欢、王飚、宁立君、赵志波、贺健、魏冰、葛涵清、李建伟等。

由于作者水平有限，加之时间仓促，书中难免出现不足之处，请各位读者评批指正。如果有任何疑问，欢迎访问http://blog.sina.com.cn/shuji，或者发邮件到deegua@126.com。

姜　波

2008年11月于北京

目 录

前 言

第一部分 Windows Embedded CE 6.0程序设计基础

第1章 嵌入式系统简介 ……2
1.1 嵌入式系统的特点和分类 ……2
1.1.1 嵌入式系统的特点 ……2
1.1.2 嵌入式系统的分类 ……3
1.2 嵌入式系统的硬件组成 ……4
1.2.1 嵌入式系统处理器的分类 ……4
1.2.2 嵌入式微处理器的主要系列 ……5
1.2.3 嵌入式系统的存储器 ……8
1.2.4 嵌入式系统的输入方式 ……10
1.2.5 嵌入式系统的输入/输出端口 ……11
1.2.6 时钟震荡电路和时钟单元 ……11
1.2.7 中断控制器 ……11
1.2.8 总线 ……11
1.3 嵌入式系统软件的特点、分类和开发 ……14
1.3.1 嵌入式系统软件的特点 ……14
1.3.2 嵌入式系统软件的分类 ……15
1.3.3 嵌入式系统软件的开发 ……17
1.4 嵌入式系统的应用领域 ……19
1.4.1 在消费类电子产品中的应用 ……19
1.4.2 在工业控制领域的应用 ……20
1.4.3 在通信领域的应用 ……20
1.4.4 在机器人、办公等其他领域的应用 ……21
1.5 嵌入式系统的发展方向 ……22
1.5.1 处理器 ……22
1.5.2 图形控制器 ……23
1.5.3 智能设备的供电解决方案 ……23
1.5.4 网络技术的应用 ……24
1.6 几种常见的嵌入式操作系统 ……25
1.6.1 嵌入式操作系统的特点 ……26

1.6.2 嵌入式操作系统的分类 ······27
1.6.3 Windows Embedded CE ······27
1.6.4 Symbian ······28
1.6.5 Palm ······29
1.6.6 嵌入式Linux ······31
1.6.7 VxWorks ······33
1.6.8 μC/OS-II ······35
第2章 嵌入式系统的软硬件协同设计 ······37
2.1 传统嵌入式系统的设计方式 ······37
2.1.1 先软件后硬件的设计方式 ······38
2.1.2 先硬件后软件的设计方式 ······38
2.2 软硬件协同设计方式 ······40
2.2.1 什么是软硬件协同设计方式 ······41
2.2.2 协同设计中处理器的选择 ······44
2.2.3 协同设计中软件平台的选择 ······46
2.3 嵌入式系统设计时需要注意的问题 ······47
第3章 Windows Embedded CE 6.0简介 ······49
3.1 Windows CE的发展历史 ······49
3.2 Windows CE 功能简介 ······53
3.3 Windows Embedded CE 6.0的新特性 ······54
3.3.1 Windows CE的不同版本 ······54
3.3.2 Windows Embedded CE 6.0的特点 ······55
3.3.3 Windows Embedded CE 6.0的体系结构 ······56
3.3.4 Windows Embedded CE 6.0的内核结构 ······57
3.3.5 Windows Embedded CE 6.0的BSP ······59
3.3.6 Windows Embedded CE 6.0的设备驱动程序 ······60
3.3.7 Windows Embedded CE 6.0对移动电话的支持 ······60
3.4 Windows Embedded CE 6.0的应用领域 ······61
3.4.1 在移动计算领域的应用 ······61
3.4.2 在嵌入式领域的应用 ······64
第4章 Windows Embedded CE 6.0开发基础 ······65
4.1 Windows Embedded CE 6.0 的开发环境 ······65
4.2 Visual Studio 2005简介 ······66
4.2.1 Visual Studio 2005的特点 ······66
4.2.2 使用Visual Studio 2005构建Smart Device工程 ······68
4.3 Platform Builder for CE 6.0简介 ······70
4.3.1 Platform Builder for CE 6.0的特点 ······70

4.3.2 使用Platform Builder for CE 6.0进行平台的定制 ……71
4.4 Embedded Visual C++ ……77
4.4.1 Embedded Visual C++简介 ……77
4.4.2 Win32 API、MFC及ATL三种程序设计方式 ……80
4.4.3 在Embedded Visual C++中构建一个MFC程序 ……80
4.4.4 在构建好的MFC程序框架中添加消息 ……83
4.4.5 EVC在安装及使用过程中应该注意的问题 ……84
4.5 ActiveSync ……84
4.6 模拟器 ……85
第5章 Windows Embedded CE 6.0中的内存管理 ……87
5.1 Windows Embedded CE 6.0中的RAM和ROM ……87
5.1.1 RAM ……87
5.1.2 ROM ……88
5.1.3 RAM和ROM在实际设备中的体现 ……88
5.2 虚存 ……89
5.2.1 Windows Embedded CE 6.0中的内存模式 ……90
5.2.2 内存状态查询函数 ……90
5.2.3 Windows Embedded CE 6.0中的分页式虚拟存储 ……92
5.2.4 与虚拟存储相关的函数 ……93
5.3 堆 ……96
5.3.1 Windows Embedded CE 6.0中堆的特性 ……96
5.3.2 本地堆 ……96
5.3.3 独立堆 ……97
5.4 栈 ……99
5.5 静态数据块 ……100
5.6 Windows CE下的Bootloader ……100
5.6.1 Bootloader简介 ……100
5.6.2 Bootloader的主要功能 ……101
5.6.3 Bootloader的结构 ……101
第6章 Windows Embedded CE 6.0中的进程和线程 ……103
6.1 Windows Embedded CE 6.0中的进程 ……103
6.1.1 进程概述 ……103
6.1.2 创建进程 ……104
6.1.3 终止进程 ……106
6.1.4 其他相关函数 ……107
6.2 Windows Embedded CE 6.0中的线程 ……108
6.2.1 线程概述 ……108
6.2.2 创建线程 ……110

6.2.3 设置线程的优先级 ……110
6.2.4 查询线程的优先级别 ……111
6.2.5 设置线程的时间片 ……111
6.2.6 挂起和恢复一个线程 ……112
6.3 进程间通信 ……112
6.3.1 事件对象 ……112
6.3.2 线程等待 ……113
6.3.3 信号量 ……115
6.3.4 互斥 ……116
6.3.5 互锁函数 ……117
6.3.6 临界区 ……118
第7章 Windows Embedded CE 6.0中的文件系统和注册表管理 ……119
7.1 Windows Embedded CE 6.0中的文件系统 ……119
7.1.1 文件系统概述 ……119
7.1.2 对象存储 ……120
7.1.3 文件的输入/输出 ……121
7.1.4 内存映射文件 ……126
7.1.5 文件管理 ……129
7.1.6 存储管理器 ……132
7.2 注册表 ……133
7.2.1 概述 ……133
7.2.2 相关API函数 ……134

第二部分 Windows Embedded CE 6.0程序设计实例

第8章 使用Platform Builder for CE 6.0进行内核定制 ……140
8.1 Platform Builder for CE 6.0简介 ……140
8.1.1 Platform Builder for CE 6.0的主要功能 ……140
8.1.2 Platform Builder for CE 6.0的目录组织 ……142
8.1.3 Platform Builder for CE 6.0的用户界面 ……142
8.1.4 Catalog Items View ……143
8.1.5 定制Windows Embedded CE 6.0平台主要包含的内容 ……144
8.2 使用Platform Builder for CE 6.0建立操作系统镜像 ……144
8.2.1 选择构建操作系统向导 ……144
8.2.2 板级支持包BSP的选择 ……145
8.2.3 模板选择 ……146
8.2.4 程序选择 ……146
8.2.5 程序开发类库的选择 ……147

8.2.6 通信服务和网络连接 ……147
8.2.7 选择操作系统的核心服务 ……148
8.2.8 设备管理 ……149
8.2.9 选择文件系统和数据存储方式 ……149
8.2.10 字体的选择 ……150
8.2.11 图形和多媒体技术的支持 ……150
8.2.12 选择所支持的国家 ……150
8.2.13 选择客户端的Internet服务 ……151
8.2.14 安全属性的选择 ……152
8.2.15 选择用户界面 ……152
8.2.16 IP语音电话选择 ……152
8.2.17 错误报告机制的选择 ……152
8.3 生成操作系统镜像 ……153
8.4 运行生成的操作系统镜像 ……154
8.5 操作系统镜像SDK的导出和安装 ……156
8.5.1 SDK的添加 ……156
8.5.2 SDK的编译 ……158
8.5.3 SDK的安装 ……158

第9章 Hello Windows Embedded CE 6.0 ……160

9.1 在命令行模式下构建“Hello Windows Embedded CE 6.0程序” ……160
9.2 在自定义模拟器上构建“Hello Windows Embedded CE 6.0程序” ……162

第10章 在Windows Embedded CE 6.0设备中实现图形的绘制 ……166

10.1 文本显示相关的函数 ……166
10.1.1 DrawText函数 ……166
10.1.2 ExtTextOut函数 ……167
10.1.3 文本的属性 ……167
10.1.4 SetBkColor函数 ……168
10.1.5 SetBkMode函数 ……168
10.2 Windows Embedded CE 6.0中的字体 ……169
10.3 图形绘制 ……170
10.3.1 直线的绘制 ……170
10.3.2 画笔 ……171
10.3.3 刷子 ……172
10.3.4 矩形 ……173
10.3.5 圆 ……173
10.3.6 圆角矩形 ……174
10.4 使用相关API函数在Windows Embedded CE 6.0中进行屏幕绘制 ……174
10.4.1 程序实例中涉及的函数框架 ……174

10.4.2 程序实例中涉及的文本显示函数 ……176
10.4.3 程序实例中涉及的图形绘制函数 ……177
10.4.4 程序实例运行效果 ……178
10.4.5 程序源代码——TextDraw.h ……178
10.4.6 程序源代码——TextDraw.cpp ……179

第11章 Windows Embedded CE 6.0中的输入处理 ……184

11.1 输入处理概述 ……184
11.2 键盘输入方式 ……184
11.2.1 输入焦点 ……185
11.2.2 系统的键盘消息 ……185
11.2.3 键盘测试 ……188
11.3 系统中鼠标的输入 ……189
11.4 Windows Embedded CE 6.0中的触摸屏 ……190
11.4.1 手写笔消息 ……190
11.4.2 电子墨水 ……190
11.4.3 关于输入焦点 ……191
11.4.4 右键单击 ……191
11.5 在Windows Embedded CE 6.0中实现鼠标点击绘制 ……192
11.5.1 程序实例中涉及的主要函数 ……192
11.5.2 程序运行效果 ……193
11.5.3 程序源代码——ClickDraw.h ……193
11.5.4 程序源代码——ClickDraw.cpp ……194
11.6 在Windows Embedded CE 6.0中实现鼠标移动轨迹的绘制 ……200
11.6.1 程序实例涉及的主要函数 ……201
11.6.2 程序运行效果 ……202
11.6.3 程序源代码——PenTest.h ……202
11.6.4 程序源代码——PenTset.cpp ……203

第12章 Windows Embedded CE 6.0版“连连看”游戏程序的设计 ……207

12.1 面向对象程序设计简介 ……207
12.1.1 传统软件设计方法存在的问题 ……207
12.1.2 面向对象程序方法的特点 ……208
12.1.3 面向对象程序设计方法中涉及的基本概念 ……208
12.1.4 面向对象程序设计方法中对象的概念 ……210
12.1.5 面向对象程序设计方法中类的概念 ……210
12.1.6 面向对象中的继承 ……210
12.2 “连连看”游戏程序介绍 ……211
12.3 “连连看”游戏程序的设计步骤 ……212
12.3.1 程序工程的构建 ……212

12.3.2 程序画面的添加 ……214
12.3.3 程序代码的添加 ……214
12.4 基类Object源代码详解 ……215
12.4.1 成员变量的含义 ……215
12.4.2 关于基类的构造函数 ……216
12.4.3 基类中其余函数代码的解释 ……216
12.5 卡片类Card源代码详解 ……219
12.5.1 成员变量的含义 ……219
12.5.2 关于卡片类的构造函数 ……219
12.5.3 卡片类中其余函数代码的解释 ……220
12.6 卡片集合类Cards源代码详解 ……220
12.6.1 成员变量的含义 ……220
12.6.2 关于卡片集合类的构造函数 ……221
12.6.3 与卡片处理、绘制、重置相关的函数 ……222
12.6.4 游戏程序中对屏幕点击的处理 ……223
12.6.5 判断两张卡片之间是否可以连通 ……224
12.7 游戏类Program源代码详解 ……229
12.7.1 成员变量的含义 ……229
12.7.2 游戏程序的初始化 ……230
12.7.3 游戏对象的处理 ……232
12.7.4 游戏程序的主函数 ……233
第13章 使用Visual Basic设计Windows Embedded CE 6.0版俄罗斯方块 ……234
13.1 程序构建过程 ……234
13.2 源代码详解 ……236
13.2.1 全局量的声明 ……236
13.2.2 相关按键和菜单的处理 ……237
13.2.3 方块的下落 ……238
13.2.4 游戏的初始化 ……238
13.2.5 游戏的重置 ……238
13.2.6 游戏开始 ……239
13.2.7 游戏的暂停和退出 ……239
13.2.8 游戏数据的加载 ……239
13.2.9 方块的加载 ……240
13.2.10 方块的放置 ……240
13.2.11 方块的旋转 ……241
13.2.12 方块的移动和落定函数 ……242
13.2.13 判断移除整行 ……243
13.2.14 程序画面的实现 ……244

13.2.15 程序画面 ……246

第14章 使用Embedded Visual C++进行Windows CE程序设计 ……247

14.1 Embedded Visual C++介绍 ……247

14.1.1 EVC4.0特性简介 ……247

14.1.2 EVC中的模拟器 ……250

14.1.3 EVC中的远程调试工具 ……251

14.2 用EVC编写俄罗斯方块 ……254

14.2.1 MFC程序设计和Win32程序设计的不同 ……254

14.2.2 程序简介 ……255

14.2.3 工程的建立 ……255

14.3 游戏程序界面的设计 ……257

14.4 源代码解释 ……260

14.4.1 fangkuaiDlg.h代码注释 ……260

14.4.2 fangkuaiDlg.cpp代码注释 ……261

第三部分 Windows Embedded CE 6.0高级程序设计

第15章 Windows Embedded CE 6.0中的中断处理 ……280

15.1 中断基础知识 ……280

15.1.1 汇编语言与微处理器的体系结构 ……280

15.1.2 中断处理程序 ……283

15.1.3 保存和恢复上下文 ……283

15.1.4 中断禁止和中断延迟 ……284

15.1.5 共享数据问题的产生 ……285

15.1.6 共享数据问题的解决 ……288

15.2 Windows Embedded CE 6.0中断的组成 ……288

15.2.1 中断服务例程（ISR） ……289

15.2.2 中断服务线程（IST） ……290

15.3 Windows Embedded CE 6.0中断处理机制的特点 ……291

15.4 Windows Embedded CE 6.0中断发生和处理的过程 ……291

第16章 Windows Embedded CE 6.0中的驱动程序 ……293

16.1 Windows Embedded CE 6.0中的驱动程序模型 ……293

16.1.1 什么是驱动程序 ……293

16.1.2 本机驱动程序 ……293

16.1.3 流接口驱动程序 ……293

16.1.4 Windows Embedded CE 6.0中的驱动程序接口 ……294

16.2 驱动程序的组成 ……295

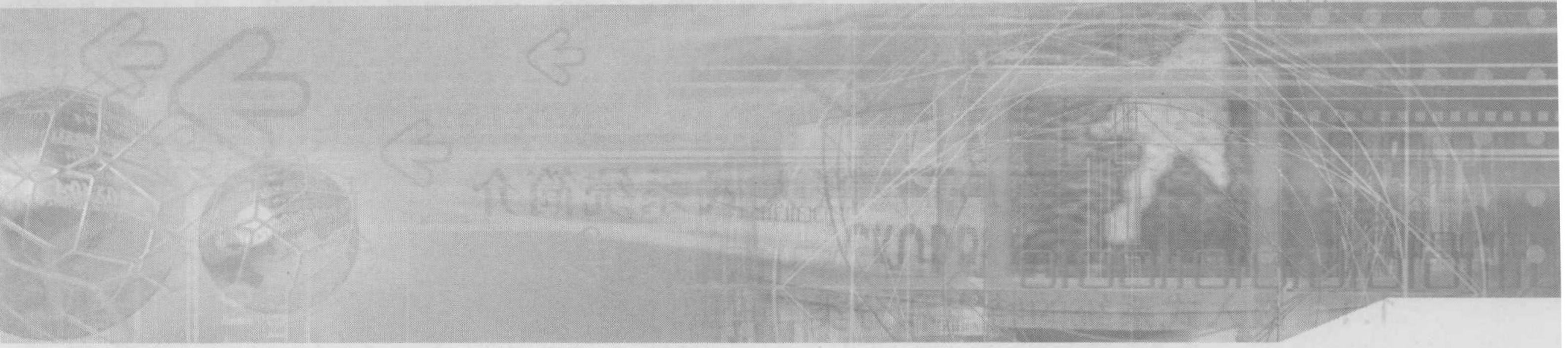
16.2.1 单体驱动程序 ……296

16.2.2 分层驱动程序 ……296
16.2.3 单体/分层驱动程序在Windows Embedded CE 6.0中的集成 ……296

第17章 Windows Embedded CE 6.0中的流驱动程序和注册表项 ……298
17.1 流驱动程序简介 ……298
17.1.1 什么是流接口驱动程序 ……298
17.1.2 流接口驱动程序的系统结构 ……299
17.2 设备文件名 ……300
17.2.1 设备文件名的格式 ……300
17.2.2 设备文件名的前缀 ……300
17.2.3 设备文件名的索引 ……300
17.3 流驱动的标准接口 ……301
17.3.1 常用的接口函数 ……301
17.3.2 接口函数的函数原型 ……301
17.3.3 与应用程序之间的调用关系 ……304
17.3.4 文件操作和设备操作函数参数间的对应关系 ……304
17.3.5 单访问和多访问 ……306
17.4 样本流接口驱动程序 ……307
17.5 Windows Embedded CE 6.0的注册表 ……307
17.5.1 Windows Embedded CE 6.0注册表的结构 ……308
17.5.2 Windows Embedded CE 6.0的常用键值类型 ……308
17.5.3 流驱动程序用到注册表设置 ……309
17.5.4 在驱动程序中访问注册表 ……310
17.6 流接口驱动程序的加载和卸载 ……312
17.6.1 流接口驱动程序的加载 ……312
17.6.2 流接口驱动程序的卸载 ……313

第18章 Windows Embedded CE 6.0中的总线 ……314
18.1 总线概述 ……314
18.2 总线的分类和主要参数 ……315
18.2.1 总线的分类 ……315
18.2.2 总线的主要参数 ……316
18.3 内部集成总线 ……316
18.3.1 内部集成总线的特点 ……316
18.3.2 传输方式 ……317
18.3.3 内部集成总线与硬件设备之间的数据交换 ……318
18.4 PCI总线 ……320
18.4.1 PCI总线的主要性能及特点 ……320
18.4.2 PCI总线的外设自动识别 ……321
18.4.3 Windows CE 6.0中PCI总线驱动的作用 ……321

18.4.4 PCIBus.dll对PCI驱动的加载 ······321
18.5 USB总线 ······322
18.5.1 USB结构的特点 ······322
18.5.2 USB设备的结构 ······323
18.5.3 USB设备的传输类型 ······324
18.5.4 USB设备的软件设计 ······324
18.5.5 Windows CE 6.0下USB系统的组成 ······324
18.5.6 Windows Embedded CE 6.0下USB设备驱动程序的编写 ······325



第一部分

Windows Embedded CE 6.0

程序设计基础

第1章　嵌入式系统简介

1.1　嵌入式系统的特点和分类

在IT行业，如果把公元2000年作为科技史的一个分水岭，那么2000年之前可以称之为PC时代；而2000年之后则要称为“后PC”时代。这是因为在“后PC”时代，多种多样的嵌入式产品已经开始逐渐取代PC，成为人们处理信息的工具。从大容量MP3、时尚的ipod Vedio、ipod nano、1000万像素的小型数码相机等消费类电子产品，到各种品牌多种功能的MP4播放器、GPS导航仪，再到机顶盒、网络家电、工业机器人及工控产品……嵌入式技术在不知不觉地改变着我们的生活。巨大的市场前景催生了嵌入式这一充满商机的产业。

在我们的生活中，很多东西都是依靠嵌入式计算机而运行的。现代生活的各个领域，从日常生活用品到交通工具等都有嵌入式计算机的身影。例如，手机就是一个典型的嵌入式计算设备，它里面包含了处理器、各种应用软件以实现通信等相关功能。另外数码相机、日常乘坐的汽车以及各种工业控制装置中都“嵌入”了各种嵌入式系统。根据IEEE的定义，嵌入式系统是“控制、监视或者辅助设备、机器、运行和车间运行的装置”。通俗地讲，嵌入式系统就是以应用为中心、以计算机技术为基础的专用计算机系统。它是与我们经常提到的桌面PC、小型机和大型机相对而言的。通常情况下，嵌入式系统不以独立设备的形式出现，其大部分或者全部系统都嵌入在其他设备中，根据主体设备的需要来发挥自己的作用。

从20世纪70年代单片机的出现到今天各种嵌入式设备的大规模应用，嵌入式系统已经有了30多年的历史。与常用的桌面PC系统一样，嵌入式系统的发展也是伴随着嵌入式软件和硬件的发展而迅速壮大起来的。从简单的4位、8位控制芯片到32位处理器，从单机系统到分布式系统，嵌入式软件的结构越来越复杂，嵌入式硬件的功能也越来越强大。这里提到的嵌入式系统，最早是基于单片机的。20世纪70年代，单片机的大规模应用使得家电制造、工业设备以及其他各种商品开始普遍使用内置的电子装置。当时那些内置了电子装置的设备已经具备了嵌入式系统的一些特点。

进入到20世纪80年代，一些嵌入式系统的程序设计人员开始进入到嵌入式软件的编写领域，这就大大地提高了嵌入式软件的开发效率，同时也促进了嵌入式硬件设备的发展。跨入到20世纪90年代以后，随着各行业对实时设备的需求越来越大，实时性的嵌入式操作系统被应用到越来越多的领域。此时，越来越多的公司看到了嵌入式系统良好的市场前景，纷纷加入到这个领域中来。市场上出现了Palm、Windows CE、嵌入式Linux等多种嵌入式操作系统。嵌入式系统正式进入到高速发展时期。

1.1.1　嵌入式系统的特点

前面已经介绍了嵌入式系统的定义，下面来看一下嵌入式系统具有哪些特点。

- 嵌入式系统在通常情况下是面向特定的用户和特定的行业的。因此其设计和开发必须和行

业特点结合在一起。

- 嵌入式系统的CPU等硬件设备必须具有体积小、功耗低、集成度高的特性，只有这样才能把很多功能集成在一个嵌入式芯片的内部。
- 嵌入式系统是计算机技术、半导体技术以及电子技术等与各个行业特点和行业应用结合在一起的产物。
- 嵌入式系统的软件和硬件都具有高效、简洁的特性。同时由于嵌入式设备的硬件性能有限，因此需要对嵌入式设备的软件和硬件进行一定的裁剪，以便做到量体裁衣。
- 嵌入式系统具有较高的可靠性、较低的成本和较低的耗电量。在一些特殊的领域中，实时性是对嵌入式系统的基本要求。
- 嵌入式系统由于其自身的特殊性，需要特殊的开发工具和程序设计环境。本书使用的开发工具有Embedded Visual C、Visual Studio 2005、Platform Builder for CE 6.0，使用的程序设计语言有C、C#和Visual Basic。
- 嵌入式系统在通常情况下是和具体的产品结合在一起的。因此嵌入式系统的升级换代往往是伴随着所附着产品的升级换代而完成的。即使同一种产品，只要产品的硬件指标不同（例如，有的芯片是ARM，有的是Intel的XScale），一般都需要对软件进行相应的移植。

1.1.2　嵌入式系统的分类

按照是否作为独立单元进行工作、是否具备实时操作、是否具备网络连接功能等方面，嵌入式系统可分为下面4种类型。

1）单机嵌入式系统。单机嵌入式系统是以非网络方式进行工作的。这种嵌入式系统的输入可以是一个按钮命令，可以是一个脉冲信号，也可以是来自前一个设备的指令信号。而它的输出可以是用户显示信息，也可以是一个电信号。通常情况下，单机嵌入式系统被应用在生产线流水作业中。

2）网络设备。和单机嵌入式系统相对的是有一些嵌入式设备是和网络连接在一起的。网络可以是公司内部的局域网，也可以是Internet互联网。这些系统能够监测一些指定的参数，同时会通过网络把这些监测到的参数信息（如当地的风速、空气的温度、大气的压力、空气的湿度等信息）传送到远方的指挥中心或者信息处理中心，以便实现实时在线监控。

3）实时的嵌入式系统。在上面介绍的网络设备中，有一些嵌入式系统可以通过网络把采集来的数据及时地送回到中央数据处理区，以方便随时对数据进行采集和监控。在这里就涉及另外一种嵌入式系统——实时嵌入式系统。这种系统要求设备在特定的时间内完成指定的任务。例如，在石油行业应用较多的扭矩测量仪，当测量仪实时测得的扭矩值大于某个指定数值的时候，设备便会自动发出指令，停止外部机械手臂的操作，以使得扭矩恰好在符合要求的范围内。

4）移动型嵌入式系统。大多数网络设备和实时嵌入式系统依靠有线网络来传输数据。随着无线技术的迅速发展，以及无线网络设备的广泛应用，越来越多的嵌入式系统开始使用无线方式进行数据传输。移动型嵌入式设备可以随时随地地实现互联网接入、收发电子邮件等功能，大大扩展了嵌入式系统的应用领域和应用范围。

许多移动型嵌入式系统（设备）同时还具备应用程序下载功能，可以根据不同的应用场合来选择相应的应用软件，极大地提高了嵌入式系统的通用性，在一定程度上也降低了嵌入式设备的使用成本。

1.2　嵌入式系统的硬件组成

嵌入式系统一般由处理器、存储器、I/O设备和相应的软件产品组成。这里主要介绍嵌入式系统的处理器、存储器等和硬件相关的内容。首先来看嵌入式系统的处理器。

1.2.1　嵌入式系统处理器的分类

在嵌入式系统领域，嵌入式处理器的内核几乎全是RISC指令集的内核（RISC的全称是Reduced Instruction Set Computer，即精简指令集计算机）。RISC是在1979年由美国加州大学伯克利分校提出的，它的体系结构具有如下一些特点：

- 采用固定长度的指令格式。
- 基本寻址方式有两三种。
- 使用单周期指令，便于流水线操作执行。
- 大量使用寄存器，数据的处理指令只针对寄存器进行，有效地提高了指令的执行效率。

RISC结构体系有两大主流：硅谷图形公司的MIPS技术和ARM公司的 Advanced RISC Machines技术。从整个微处理器领域来讲，RISC和 CISC（复杂指令集计算机）是目前设计制造微处理器的两种典型技术。虽然它们都试图在体系结构、操作运行、软件硬件和编译时间等诸多因素中做出某种平衡，以求达到高效的目的，但由于各自采用的方法不同，它们在很多方面差异很大。它们的主要区别体现在以下7个方面：

1）指令系统。RISC的设计者把主要精力放在经常使用的指令上，尽量使它们具有简单高效的特色。对于不常用的功能，常通过组合指令来完成。因此，在RISC上实现特殊功能时，效率可能较低。但可以利用流水技术和超标量技术来加以改进和弥补。而CISC的指令系统比较丰富，有专用指令来完成特定的功能。因此在处理特殊任务的时候效率较高。

2）存储器操作。RISC对存储器的操作是有限制的；而CISC由于机器存储器的操作指令较多，因此操作较为直接。

3）中断。RISC在指令执行的适当地方可以响应中断，而CISC在指令执行结束后才可以响应中断。

4）CPU。RISC的CPU包含较少的电路单元，因而面积小、功耗低；而 CISC的 CPU包含丰富的电路单元，因而功能强、面积大、功耗大。

5）设计周期。RISC微处理器结构简单，布局紧凑，设计周期短，易于采用最新技术；CISC微处理器结构复杂，设计周期长。

6）程序。RISC汇编语言程序一般需要较大的内存空间，实现特殊功能时程序复杂；而CISC汇编语言程序编程相对简单，在科学计算和复杂操作的程序设计上相对容易。

7）应用范围。由于RISC指令系统与特定的应用领域有关，故RISC更适合于专用机；而CISC则更适合于通用机。

表1-1展示了RISC和CISC指令集在不同微处理器中的应用。

表1-1　RISC和CISC指令集在不同微处理器中的应用

公司名称	系　列	RISC	CISC
Intel	8086		✓
	i860	✓	
IBM	Power PC 604,601	✓	
Motorola	68HCxxx	✓	
	MPC620	✓	
Sun	SPARC	✓	

目前嵌入式系统的处理器可以分为下面4类。

（1）嵌入式微处理器

嵌入式微处理器（Micro Processor Unit，MPU）是由通用计算机的CPU演变而来的。和常用的PC处理器不同，在嵌入式领域中，只保留了和嵌入式应用紧密相关的功能，去掉了其他冗余的功能，实现了低功耗和高资源利用率的结合。嵌入式处理器具有体积小、质量轻、成本较低及可靠性较高的特点。经常提到的ARM等都属于嵌入式微处理器。常用的嵌入式微处理器主要有ARM系列、MIPS系列、X86系列、Motorola系列等。

（2）嵌入式微控制器

在工业控制领域中，经常提到的单片机就属于嵌入式微控制器（Micro Controller Unit，MCU）范畴。尽管单片机最早出现在20世纪70年代，但是经过了近30年的发展，这种8位的电子器件在嵌入式设备中仍然有着非常广泛的应用。和嵌入式微处理器不同，嵌入式微控制器的体积大大减小，并且功耗和成本也有所下降。目前，嵌入式微控制器已经成为嵌入式工业控制中的主流。

（3）嵌入式DSP处理器

熟悉嵌入式软件设计的读者一定会经常看到DSP这个词，这里提到的DSP（Digital Signal Processor）是专门用来进行信号处理的处理器。它在系统结构和指令算法上进行了新的特殊的设计，因此具备了较高的编译效率和较快的指令执行速度。DSP在数字滤波和频谱分析等各种仪器上得到了广泛的应用。世界上首枚DSP芯片诞生在1982年，其运算速度比嵌入式微处理器快了几十倍，因此在语音合成和编码解码中得到了非常广泛的应用。目前市场上流行的MP3播放器就是DSP技术一个很好的体现。

（4）嵌入式片上系统

嵌入式片上系统（System On Chip，SOC）是目前嵌入式领域比较热门的话题之一。它最大的特点在于实现了软件和硬件的无缝集合，在单块硅片上集成了处理器、高密度逻辑电路、模拟和混合信号电路、存储器和通信电路等。随着其应用领域的日趋广泛，SOC在声音、图像、网络应用等领域发挥着越来越重要的作用。

1.2.2　嵌入式微处理器的主要系列

微处理器是嵌入式系统的核心部分，通常会包含以下三个组成部分：

- 控制单元。
- 算数逻辑单元。
- 寄存器。

图1-1展示了嵌入式微处理器的组成。

目前在嵌入式领域，主要有以下4个系列的嵌入式微处理器。

1. ARM系列

ARM公司成立于1991年，它本身并不生产芯片，而是采取出售芯片设计授权的方式扩大其影响力。通常情况下，世界各大半导体生产厂商从ARM公司购买ARM核，然后根据各自不同的需要，针对不同的应用领域添加适当的外围电路，从而生产出自己的ARM微处理器芯片。目前，ARM微处理器已经在工业控制、消费类电子产品、通信设备、网络设备以及无线技术等领域得到了极为广泛的应用。而基于ARM技术的微处理器应用大约占到了32位RISC处理器3/4以上的市场份额。在可预见的时间内，ARM微处理器仍将在嵌入式领域占据主导地位，将会有越来越多的产

品应用这种设计，ARM微处理器也将会对我们的生活产生日益广泛的影响。

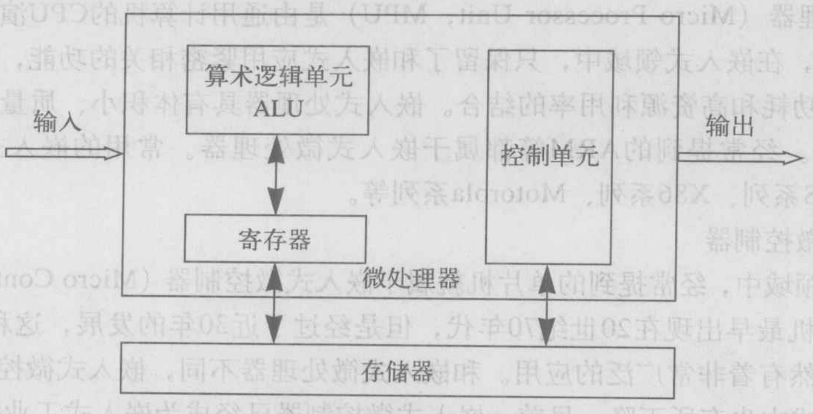


图1-1 嵌入式微处理器的组成

ARM嵌入式微处理器之所以可以占据如此之大的市场份额，得益于ARM公司使用的良好的基础体系结构，在功耗、数据传输效率、数据处理速度以及DSP功能方面都处于业界领先的地位。日常生活中经常使用的手机里，以及目前日益流行的手持播放器、数码相机等各种消费类电子产品中都有ARM的身影。

目前，基于ARM架构的处理器主要应用在以下几方面。

- 工业控制领域：作为32位的RISC架构，基于ARM核的微控制器芯片不但占据了高端微控制器市场的大部分市场份额，同时也逐渐向低端微控制器应用领域扩展。ARM微控制器低功耗、高性价比，已经向传统的8位、16位甚至32位微控制器提出了挑战。
- 无线通信领域：目前已有超过85%的无线通信设备采用了ARM技术，ARM以其高性能和低成本，在越来越多的通信设备上得到了应用。
- 网络应用：随着宽带技术的推广，采用ARM技术的ADSL芯片正逐步获得竞争优势。此外，ARM在语音及视频处理技术上进行了优化，并获得了广泛的支持。
- 消费类电子产品：ARM技术被目前流行的数字音频播放器、数字机顶盒和游戏机广泛采用。
- 成像和安全产品：现在流行的数码相机绝大部分采用ARM技术。手机中的32位SIM智能卡也采用了ARM技术。

图1-2展示了采用ARM内核的诺基亚N95手机。

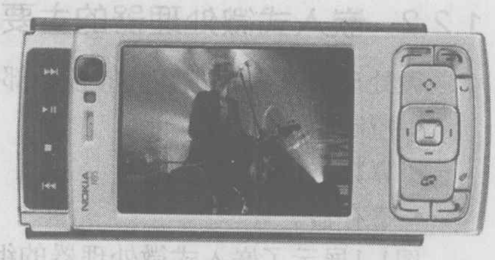


图1-2 采用ARM内核的诺基亚N95手机

2. X86系列

提到X86系列，广大的计算机用户一定不会陌生。因为X86结构已经主宰了PC市场几乎100%的市场份额。我们经常使用的台式机和笔记本电脑都是X86结构的。近年来，由于PC处理器市场的竞争日益激烈（CPU的价格已经相继几次降价，并且双核、多核处理器也进入到普通PC市场），因此X86处理器也开始向嵌入式领域转移。那么作为一个在传统PC领域有强大号召力的芯片结构，在嵌入式领域是否具有得天独厚的优势呢？

熟悉嵌入式开发的读者一定知道，对于嵌入式领域的开发，不同领域的嵌入式产品设计会有不同的要求。例如，电信领域的工程师在进行嵌入式产品设计时，由于产品的设计周期和产品的

寿命较长，他们更倾向于选择那些稳定性高、可以长期提供货源、有成熟开发工具的处理器，因此一个拥有较大市场份额、技术成熟、为用户所熟知的处理器产品就更具有吸引力；而对于从事网络产品设计和开发的工程师来说，产品的升级换代较快，因此那些技术先进、开发周期较短、同时具有单一架构的处理器产品则更具吸引力。可以说，几乎没有符合所有设计要求的嵌入式芯片。尽管如此，X86结构作为一种性价比较高、使用人数众多、技术成熟、产品线完善的产品还是越来越受到广大嵌入式产品设计师的青睐。

图1-3展示了一种嵌入式X86计算机主板。

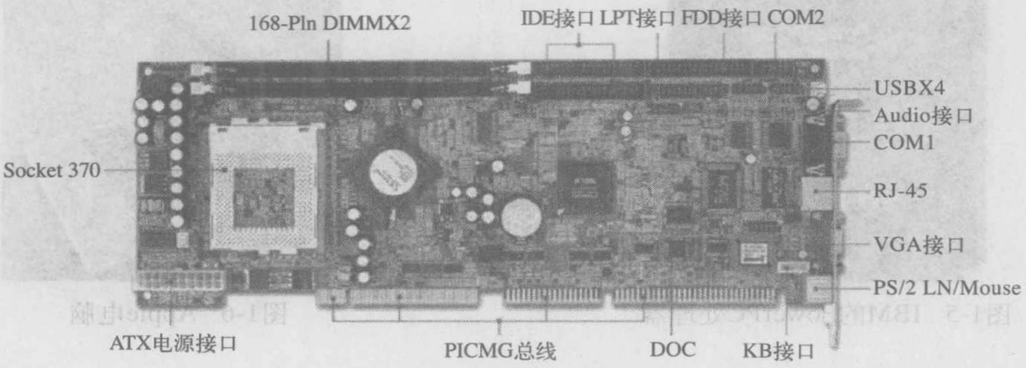


图1-3 嵌入式X86计算机主板

3. MIPS系列

MIPS公司也是一家从事嵌入式处理器设计的厂商。它主要设计高性能的32位和64位处理器。该公司成立于1984年，它设计的RISC处理器最早始于20世纪80年代初期。在1999年，MIPS公司发布了MIPS32和MIPS64架构标准，这就为未来的MIPS处理器奠定了基础。

目前，基于MIPS的设计已经集成在全球成千上万的产品中。包括Linksys的宽带设备、佳能的数码相机、索尼的数字电视和娱乐系统（Play Station）、先锋（Pioneer）的 DVD刻录机、摩托罗拉的数字机顶盒、思科的网络路由器和惠普的激光打印机。虽然MIPS可以提供32位和64位的CPU架构，但MIPS关注的却并不是所谓的通用PC和服务器市场。

从技术创新上看，多线程技术一直是MIPS在高性能CPU方面的主要发展方向。因为单线程微处理器在等待存储器存取时浪费了很多周期，在很大程度上限制了系统性能。而多线程技术从根本上解决了这个问题。我们可以通过增加处理器的利用率来减少存储器等待的时间。当一个线程停止时，其他线程立刻进入流水线并开始执行，使应用吞吐能力显著增加。目前，MIPS已经能够提供单线程处理器和多线程处理器在系统中有效地混合应用。

图1-4展示的是采用了MIPS芯片的PlayStation游戏机。

图1-4 索尼PlayStation游戏机

4. Motorola系列

Motorola系列嵌入式处理器主要包括68K/ColdFire系列和PowerPC系列。这里主要介绍PowerPC系列。PowerPC系列是由IBM、Apple和Motorola公司共同研制和开发的，这种架构的特点是可伸缩性良好、方便灵活。同时，该体系结构一个非常大的优点就是开放性。在这种架构中，

定义了一个指令集，同时允许任何设计人员来设计和修改与PowerPC兼容的处理器。在此基础上开发的支持PowerPC软件的源代码都是可以自由使用的。

PowerPC内核被摩托罗拉公司广泛地应用于嵌入式领域，已经成为通信领域中使用最为广泛的处理器内核。国内知名的中兴通信、华为等都在各自的通信产品中采用了摩托罗拉的PowerPC嵌入式处理器。

图1-5展示的是IBM的PowerPC处理器，图1-6展示的采用该处理器的苹果电脑。

图1-5 IBM的PowerPC处理器

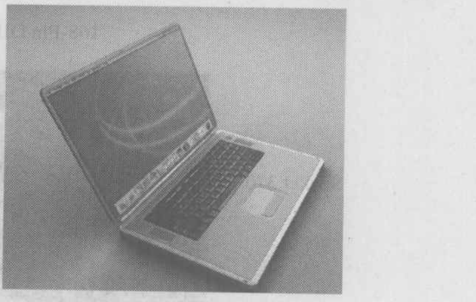


图1-6 Apple电脑

1.2.3 嵌入式系统的存储器

我们知道，计算机中的存储器是存储数据和程序的部件系统，通常情况下存储器是由地址译码器和存储单元构成的。常用的存储器有RAM、ROM、EPROM、EEPROM和闪存。表1-2列出了常用存储器的分类。

表1-2 常用存储器的分类及其单元构造

种类		存储器单元构造
ROM	EEPROM	2晶体管+隧道区域
	EPROM	1晶体管
	FLASH	1晶体管
	OTP	1晶体管
	MASK ROM	1晶体管
RAM	DRAM	1晶体管+电容器
	SRAM	4晶体管+2负载元件
	NVRAM	SRAM单元+EEPROM单元

下面来看一下上面提到的各种存储器都有什么样的特点。

RAM的英文全称是Random Access Memory，即随机访问存储器。在系统运行时，处理器会将需要处理的数据写入到RAM中临时存储。需要特别提及的是，在RAM中存储的信息需要电力来维持。也就是说，当系统掉电以后，存储在RAM中的数据将会全部丢失。RAM通常分为静态RAM和动态RAM。其中静态RAM又被称为SRAM，动态RAM又被称为DRAM。SRAM是RAM中速度最快的一种存储器，它需要较少的外围电路的支持，具有相对较低的功耗。而它的缺点则是容量比DRAM小，一次在同样存储容量的条件下就需要相对较多的存储芯片，因此成本相对较高。在这里有一点需要特别指出的是，在Pocket PC中，RAM被划分成了两个部分，一个部分负责软

件等数据的存储，一个部分负责存储程序运行时候需要的数据。二者之间空间容量的大小是可以手工调整的，同时根据软件对系统资源占有的不同，Pocket PC等智能设备也可以根据实际情况自动对二者的大小进行调整。

而DRAM是用电容器阵列来实现对每一个数据的保存的。大家都知道电容器中的电荷是可以短暂保存一段时间的，因此在电荷消失之前，存储在DRAM中的数据还能被短时间地保存。正因如此，在实际应用中我们需要不断地对DRAM进行刷新，而这种刷新机制是需要额外电路来支持的，同时还会对处理器的访问造成一段时间的延迟。

ROM的英文全称是Read Only Memory，即只读存储器。该种存储器属于非易失性存储器，也就是说存储在ROM中的数据是不易丢失的。在一个嵌入式设备的系统中，通常情况下需要把系统启动时的一些信息以及操作系统的信息存储在ROM中，来实现对硬件设备的初始化。

下面以Pocket PC为例来说明ROM和RAM的作用。当我们拿到一个已经预装操作系统的Pocket PC以后，里面的操作系统（如Pocket PC2003等）是被保存在ROM中的。操作系统自带的那些软件如Pocket Word、Pocket Excel等都和操作系统一样，被保存在ROM中。而我们后来自定义安装的软件，如金山词霸、极品飞车游戏等则被存储在RAM当中。说到这里，也许有的读者会问，在前面提到RAM的时候说当系统掉电以后，存储在RAM中的数据将会全部丢失，那么为什么存储在Pocket PC中的程序在关机以后，再次开启设备的时候仍旧存在呢？这是因为Pocket PC设备的“关机”和我们在桌面系统中提到的系统关机不是一个概念。在Pocket PC中，即使我们把设备关闭，屏幕看上去也不亮了，但是实际上设备自带的电池还在继续对系统供电。只不过此时设备处于“休眠”状态，消耗极少的电量来维持系统的一些数据信息。经常使用Pocket PC设备的用户一定会发现，当设备放置很长一段时间不用以后，再次开启设备的时候系统会提示重新进行初始化，而我们安装在设备中的所有自定义软件也将会全部丢失。这就是因为设备电池的电量已经全部耗光，RAM中的信息自然也就全部丢失了。而由于操作系统等是存储在ROM中的，因此当重新加电启动的时候，又会进入到系统的初始化状态中。

EPROM属于可擦写的ROM，它是通过紫外线照射来实现对数据擦除的。这种方式的缺点就是当需要对其进行重新擦写的时候，需要把芯片从电路板上取下来，擦除完毕以后还需要把它放在烧录器上进行程序的重新加载，然后再放回到电路设备中。

EEPROM则属于电擦除ROM。和EPROM不同，我们可以在电路设备上对其进行擦除和再编程。正如任何事情都没有十全十美一样，尽管EEPROM很方便实现信息的重写，但是其容量比ROM小得多，一般是用于断电情况下对系统参数、模式信息等重要数据进行保存。此种存储设备在嵌入式系统中应用得非常广泛，在网络地址存储和配置信息存储等方面都发挥着非常重要的作用。

闪存（Flash Memory）是一种新的ROM技术，它不但具有EEPROM可方便地进行数据重写的特点，还具备了ROM大容量的优势。这种存储设备的优点在于，即使在掉电的情况下，存储在其中的数据也不会丢失。同时随着技术的不断进步，ROM的成本也在不断地下降。在Pocket PC2002和Pocket PC2003等设备中，我们可以在机器的顶部插上一个扩展的SD卡，实现对数据的存储。前面提到过，存储在RAM中的数据在掉电情况下是会丢失的，因此可以通过智能设备自带的备份软件把系统数据备份到SD卡中。这样当设备重新加电启动以后，再通过备份程序把程序恢复过来即可，大大的增加了设备操作上的人性化设计，赢得了广大用户的欢迎。

目前市场上广泛使用的SD卡、MS卡和CF卡等属于闪存设备。

1.2.4　嵌入式系统的输入方式

在通常情况下，嵌入式设备不像普通计算机那样带有键盘和鼠标，取而代之的是通过触摸屏的方式来进行信息的输入。如我们经常使用的自动取款机就是通过触摸屏来进行输入的。在一些Pocket PC设备上，还可以通过系统自带的软键盘来模拟外置键盘，同时Pocket PC系统还支持对手写笔记的保存，方便在紧急情况下进行信息的记录。目前非常时尚的Tablet PC采用了手写笔输入方式，这种设备随机配备了一支手写笔，通过手写笔上的感应装置可以很清晰地捕捉到用户的笔迹。需要特殊说明的是，这里提到的Tablet PC的手写笔和Pocket PC中使用的手写笔是完全不同的。更为确切地说，Pocket PC等智能设备中使用的手写笔只是采用一个塑料和金属材质的笔来模拟鼠标点击屏幕的功能，当用手指直接点击触摸屏屏幕的时候，同样可以实现手写笔的功能。

但是Tablet PC中使用的手写笔及其采用的“数字墨水”技术则采用了电磁感应原理，不同于Pocket PC中使用的阻抗式触摸屏，手写笔不接触屏幕仍然可以在Tablet PC的屏幕上留下“痕迹”。这样不但大大地增加了触摸屏的使用寿命，也使得手写笔的使用感受更加接近人们的使用习惯。同时，用户在进行手写输入的时候，还可以把手放在触摸屏上，而不用担心在Pocket PC中经常出现的由于手指的意外触动而导致的光标移动现象的发生。

图1-7和图1-8展示了Tablet PC产品。

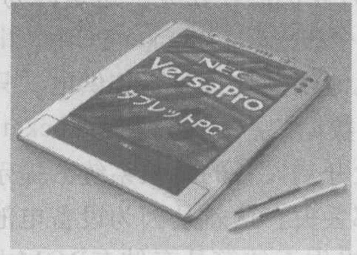


图1-7　Tablet PC (1)

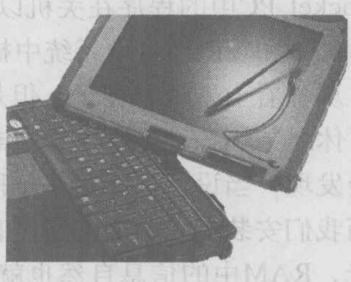


图1-8　Tablet PC (2)

在很多基于Windows CE的设备中，用户是通过手写笔和触摸屏与程序进行交互的，手写笔和触摸屏实际上就是模拟了普通PC中的鼠标操作。按下一个按钮或者单击按键的时候都会在Windows系统中产生一个消息，相应地在Windows CE系统中，当使用手写笔和触摸屏时产生的消息和鼠标产生的消息基本上是一样的。更为确切地说，在Windows CE中产生的消息是Windows消息的一个子集。当在Windows CE系统中移动或者按下手写笔的时候，就相当于单击了鼠标左键，此时系统会按照处理鼠标的方式来处理这些事件。同理，当移动手写笔的时候，系统也会像处理鼠标移动一样来处理这些事件。

例如，当在Windows Embedded CE 6.0系统的触摸屏上按下手写笔的时候，窗口会收到WM_LBUTTONDOWN消息；当从触摸屏上拿开手写笔的时候，窗口会接收到WM_LBUTTONUP消息。

因为很多嵌入式设备都没有自带的键盘，因此在它们上面进行文字的输入就存在一些问题。在Windows CE设备中，系统提供了输入面板来解决这一问题。这种输入面板的结构和我们使用的外接键盘类似，由于它是用软件来模拟的键盘，因此也被称为软键盘。

图1-9展示了Windows CE设备中的软键盘。

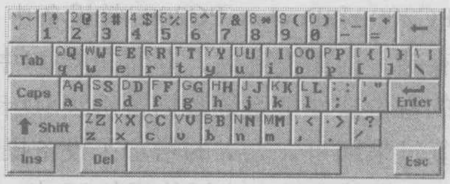


图1-9　Windows CE设备中的软键盘

此外，一些嵌入式设备中还采用了语音输入（如手机的语言拨号功能）和扫描识别输入等方式。由于本书的重点不在讲解输入方式上，因此这里就不再做更为详细的介绍了。

1.2.5 嵌入式系统的输入/输出端口

嵌入式系统的输入/输出端口主要有中断控制器、DMA、串行接口、并行接口、A/D、D/A、存储器接口、设备接口等。通过这些接口，嵌入式设备可以实现键盘、传感器、触摸屏等多种输入方式。常见的输出设备则包括LCD显示屏和发光二极管等。

嵌入式系统正是通过多种输入端口实现从物理设备获得输入，在经过系统的处理以后，如果信息数据需要输出给外界设备，则会通过输出端口输送出去。在嵌入式系统中，每一个输入/输出端口都是通过地址来标示的，系统正是通过对地址的操作来完成输入/输出功能。

1.2.6 时钟震荡电路和时钟单元

嵌入式系统的时钟主要用于控制CPU、系统定时器和CPU机器周期的各种时钟控制要求，它控制着一条指令的执行时间。例如，多处理器系统中需要一个具有较强驱动能力的时钟电路，可以同时驱动所有的处理器，因此处理单元的震荡电路需要具有高度的稳定性。

在一个嵌入式系统中，时钟输出信号为所有其他系统单元提供同步时钟。

1.2.7 中断控制器

一个嵌入式系统可能会有多个设备，系统必须为每一个设备建立相应的中断服务程序（ISR），以便用来完成控制和处理每个设备的要求。每个系统也都具有各自的中断处理机制，处理来自系统不同过程的中断。

嵌入式系统的中断具有一定的优先级，且具有多种类型。通常在中断处理开始前要保存现场数据到堆栈中，以便中断结束执行以后，系统可以返回继续执行中断发生前的程序。通常情况下，中断类型主要有以下几种：

- 复位中断。
- 未定义指令中断。
- 软件中断。
- 预取指异常中断。
- 数据异常中断。
- 一般外部中断IRQ。
- 外部快速中断FIQ。

1.2.8 总线

从微处理器的角度看，嵌入式系统的总线可以分为片外总线和片内总线。我们经常提到的ISA总线、PCI总线属于片外总线；AMBA、AVALON则属于片内总线。从实际的应用来看，我们主要使用的是片外总线。

片外总线的作用是实现模块之间、设备之间、系统之间的互联。而片外总线又可以分为并行总线和串行总线。

其中并行总线的传输速度较高，通常高达几时兆BPS，但传输距离较近，因而它主要被应用在单个设备内部模块间的互联。和并行总线恰恰相反，串行总线的传输速度较低，但是传输距离较远，因而常被用在分布式系统设备之间的互联。

下面简单介绍一下几种嵌入式系统常用的总线。

1. CPCI总线

CPCI全称为CompactPCI，中文又称紧凑型PCI，是国际PICMG协会于1994提出来的一种总线接口标准。它的出现解决了多年来电信系统工程师与设备制造商面临的棘手问题，即传统电信设备总线如VME（Versa Module Eurocard）与工业标准PCI（Peripheral Component Interconnect）总线不兼容问题。

1995年11月PCI工业计算机制造者联合会颁布了CompactPCI规范1.0版。该协会在1997年推出了CPCI 2.0规范。CPCI总线规范=PCI总线的电器规范+标准针孔连接器+欧洲卡规范。即CPCI总线继续采用PCI总线的电气规范；同时将PCI的传统机械结构改为欧洲卡结构，这样极大地改善了其散热条件，提高了它的抗震性和电磁兼容性；并将PCI的金手指互联方式改为2mm密度的针孔连接器，提高了气密性、防腐性、可靠性和负载能力。正因为CPCI总线具有的开放性、高可靠性、可热插拔性，使其被广泛应用在高恶劣环境下的工业控制、军事应用等领域。

2. PC104总线

PC104总线是工业计算机总线标准，是一种专为嵌入式控制而定义的工业控制总线。其信号定义和PC/AT基本一致，但电气和机械规范却完全不同，属于一种优化的、小型的、堆栈式结构的嵌入式控制系统。PC104总线分为以下两个版本：

- 8位。
- 16位。

PC104总线与普通PC总线控制系统的不同主要体现在：

- 小尺寸结构：标准模块的机械尺寸为3.6×3.8英寸，即96mm×90mm。
- 堆栈式连接：去掉了总线背板和插板滑道，总线以“针”和“孔”性质层叠连接，即PC104总线模块之间总线的连接是通过上层的针和下层的孔互相咬合来完成的，这种层叠封装方式具备了良好的抗震性。
- “轻便”总线驱动：PC104总线减少了元件数量和电源消耗，4mA的总线驱动就可以使模块正常工作，而每个模块仅有1~2W能耗。

3. CAN总线

CAN总线的英文名称是Controller Area Network，它是实现控制器局域网常用的现场总线之一。CAM总线的基本设计规范要求具有较高的位速率、高抗电磁干扰性，而且需要能够监测出产生的任何错误。

目前，CAN总线主要被应用在汽车电子控制系统、电梯控制系统、安全监控系统、医疗仪器、纺织机械、船舶运输等方面。

CAN总线具有以下特点：

- 成本低。
- 传输距离远，可以长达10m。
- 相对高速的数据传输速率，可以达到1Mbit/s。

• 可以根据报文的ID决定是否接收或者屏蔽该报文。

• 可靠的错误处理和检测机制。

• 当发送的信息遭到破坏后，可以自动重发。

• 节点在产生严重错误的情况下具有自行推出总线的功能。

在嵌入式系统中，实现CAN总线接口有两种方式，一种是处理器本身带有CAN总线控制器，另外一种是使用CAN总线控制芯片来实现。

图1-10展示了CAN总线控制器的体系结构。

4. SPI串行总线

SPI总线是一个4线的串行通信接口总线，用于在IC器件之间的连接。它也是一个同步的串行数据连接，支持在CPU和其他支持SPI的设备之间进行低、中带宽的网络连接。SPI总线基本上是一个简单的同步串行接口，可以用很少的信号线来连接外部低速的设备。

同时，SPI串行总线还是工作在主从模式下的。当两个设备都通过SPI总线连接时，一个作为主设备，另外一个作为从设备，主设备驱动串行时钟。并且SPI总线是全双工接口，即数据可以同时发送和接收。

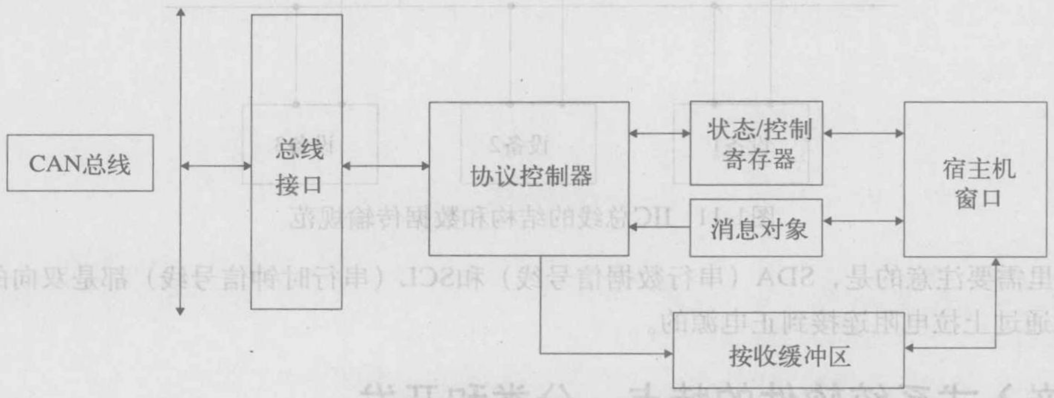


图1-10　CAN总线控制器的体系结构

SPI总线具有以下一些特点：

• 全双工，3线同步传输。

• 具有主从工作模式。

• 有1.05MHz的最大主机位速率。

• 可发送结束中断标志。

• 有写冲突保护。

• 有总线竞争保护。

5. IIC总线

IIC总线是一种简单的双向总线，是一种介于IC器件中间进行连接的二进制总线。在一些消费类电子产品、通用系统、工业控制系统中，一般都具有智能控制功能，如使用了微处理器进行控制等。而这些智能控制单元需要有通用的电路，如LCD驱动、I/O端口、RAM、EEPROM和一些数据转换电路等。通过IIC总线就可以充分利用这些设计的相似之处，使硬件的工作更有效率，电路设计更加简单。

IIC总线具有以下一些特点：

- 仅有两条总线信号线，SDA（数据信号线）和SCL（串行时钟信号线）。
- 每一个连接在此的设备是可以编址的。采用IIC总线连接的设备处于主从模式，主方既可以接收数据，也可以发送数据。
- 属于一个真正的多总线。因为包含了冲突检测和竞争机制，因此当多个主方同时发送数据时，不会造成数据冲突。
- 是串行的8位双向数据传输总线。在标准模式下，数据传输速率为100kbit/s；而在快速模式下，数据传输速率则达到了400kbit/s；在高速模式下，数据传输率则达到了3.4Mbit/s。

图1-11展示了IIC总线的结构和数据传输规范。

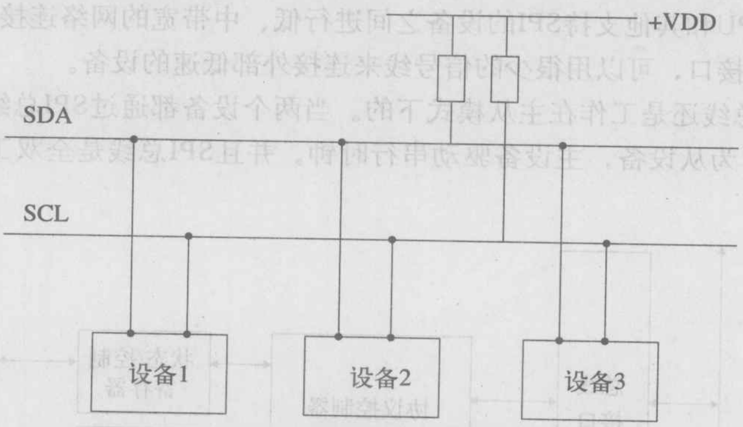


图1-11 IIC总线的结构和数据传输规范

在这里需要注意的是，SDA（串行数据信号线）和SCL（串行时钟信号线）都是双向的I/O线，它们都是通过上拉电阻连接到正电源的。

1.3 嵌入式系统软件的特点、分类和开发

嵌入式系统的软件是实现嵌入式计算机系统功能的软件。嵌入式系统的发展非常迅速，随之带动了嵌入式软件的飞速发展。嵌入式软件的发展就像生物的发展一样，经历了由单细胞到多细胞，由简单到复杂这一过程。随着嵌入式硬件性能的不断提高，以及嵌入式产品的应用日益广泛，嵌入式软件的结构也变得越来越复杂，功能也越来越强大。

1.3.1 嵌入式系统软件的特点

和传统的软件设计相比，嵌入式系统软件开发具备了一些独有的特点。

- 关于系统的响应时间：由于嵌入式系统大多应用在对实时性要求较高的场合，因此系统要能够在极短的时间内做出对事件和信号的响应。
- 关于软件的成本：在一般情况下，嵌入式系统作为整个产品的一部分，其自身成本直接关系到整个产品的成本。因此通常情况下，嵌入式软件的功能做到刚刚“够用”即可，而不像Windows应用软件功能越多越好。例如经常使用的自动取款机，显然是不需要提供多媒体播放功能的。
- 关于系统的能耗：有很多嵌入式设备都是依靠电池供电的，因此嵌入式软件要尽可能小地

消耗系统的能源，以达到节省电能的目的。

- 关于软件的可靠性：因为很多的嵌入式设备都是在无人看管的情况下运行的（很少需要我们时刻注意软件的性能，如手机，只有我们在打电话的时候才会注意到手机的信号等性能指标。而在待机状态则很少查看手机信号的强弱），因此软件运行时的可靠性和稳定性就尤为重要。尤其当嵌入式软件被应用在一些大型设备的关键部位的时候，可靠性的好坏往往会关系到整个系统的安全性、稳定性等诸多指标。
- 关于软件占用的内存空间：由于嵌入式系统的特殊性，有很多嵌入式设备的内存都非常小，例如常用的Pocket PC其内存空间大多是64MB和128MB，这样的空间大小和我们的桌面PC系统动辄几百兆字节，甚至上G字节的内存相比，空间可以称得上是少得可怜了。正是因为这样的特殊性，嵌入式软件在运行的时候不要占用过多的内存空间，要是像在Windows XP中一样，一个软件运行时会占用100多兆字节，肯定是无法运行的。
- 对CPU资源的占用：嵌入式系统的CPU频率和桌面PC相比大都很低。如Nokia的 6681手机，其CPU主频是200多MHz，而桌面PC系统的CPU早已进入到GHz的时代。因此运行在上面的嵌入式软件自然也不能占用过多的CPU资源，同时在软件的算法实现上也不能消耗大量的CPU资源。

1.3.2 嵌入式系统软件的分类

如果按照软件的结构来划分，嵌入式系统软件可以分为单线程程序、事件驱动程序等。图1-12展示了嵌入式系统软件的分类。

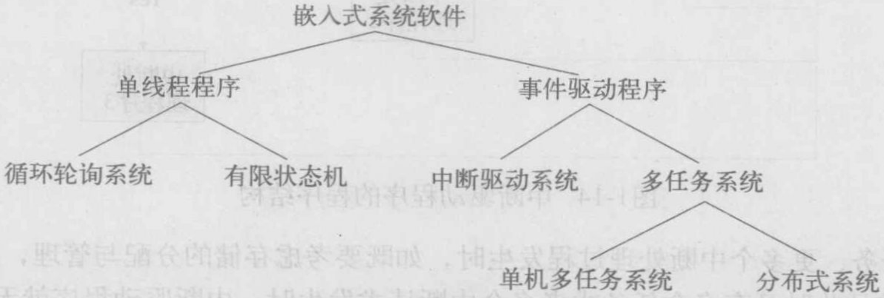


图1-12 嵌入式系统软件的分类

图1-13展示了单线程程序结构。

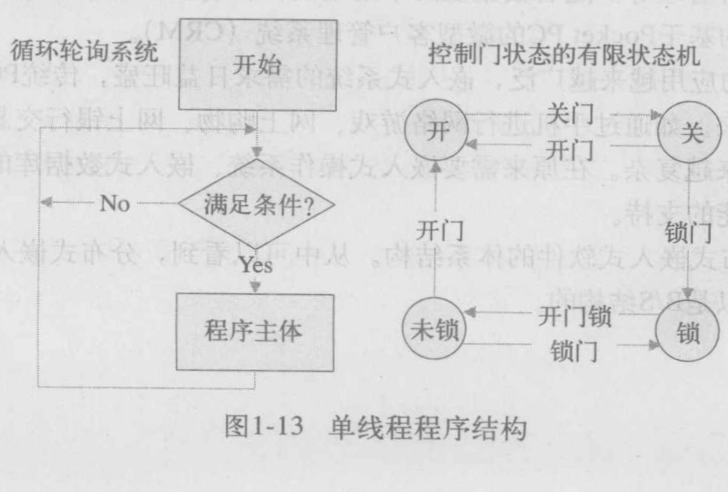


图1-13 单线程程序结构

所谓的“单线程程序”就是指没有主控程序，同时它又分为循环轮询系统和有限状态机。如一个采用循环轮询结构的产品包装系统，系统采用光感应来判断时都有产品需要包装。一旦发现传输带上有物体，程序主体就开始控制设备执行包装动作。单线程程序的优点是程序简单、执行效率高；缺点是一旦出现故障，系统便无法自动进行控制与恢复操作，安全性较差。

针对单线程程序的不利因素，又出现了事件驱动程序。这类驱动程序中的中断驱动程序可以很好地解决这种安全性问题。在中断驱动程序中，有一个循环轮询的主程序来控制中断响应程序的执行。当多个中断请求发生时，就会考虑中断请求的优先级，以及响应相应的中断请求。如果上面提到的包装系统采用了中断驱动系统，一旦系统有错误发生，就会有一个错误中断请求来处理相应的错误。处理完成以后，系统会自动恢复运行。

图1-14展示了中断驱动程序的程序结构。

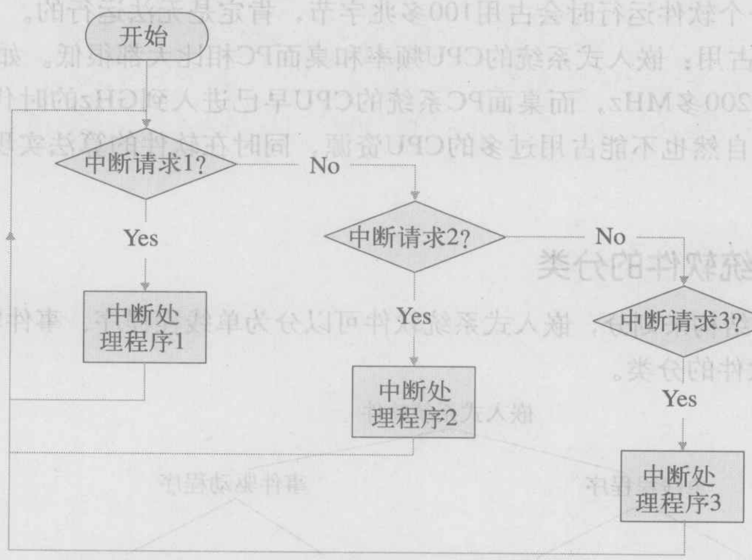


图1-14 中断驱动程序的程序结构

当更多的任务、更多个中断处理过程发生时，如既要考虑存储的分配与管理，又要考虑I/O的控制与管理，并且此时又有多个任务或者多个中断请求发生时，中断驱动程序就无法应付了，此时就需要嵌入式操作系统的支持。有了它就可以很好地完成任务之间的切换、调度、通信、同步、互斥、中断管理、时钟管理等。随着数据量的不断增长，在有些系统中还需要嵌入式数据库的支持，如目前广泛使用的基于Pocket PC的微型客户管理系统（CRM）。

随着嵌入式系统的应用越来越广泛，嵌入式系统的需求日益旺盛，传统PC上的应用几乎都会移植到嵌入式设备中来。如通过手机进行网络游戏、网上购物、网上银行交易等，这些应用都使得嵌入式软件变得越来越复杂。在原来需要嵌入式操作系统、嵌入式数据库的基础上，还需要网络通信协议等多种功能的支持。

图1-15展示了分布式嵌入式软件的体系结构。从中可以看到，分布式嵌入式软件可以是Peer-to-Peer结构的，也可以是B/S结构的。

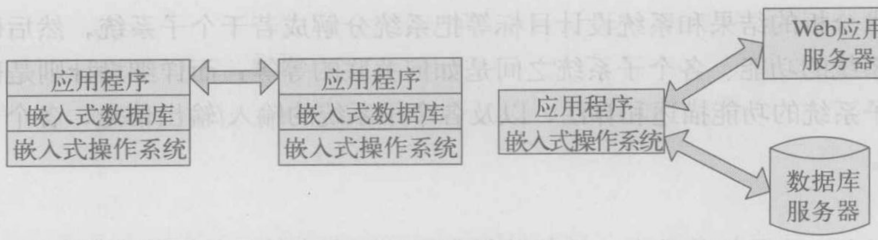


图1-15 分布式嵌入式软件的体系结构

1.3.3 嵌入式系统软件的开发

从前面的介绍中可以看到，嵌入式软件的系统结构越来越复杂，由此带来了嵌入式软件开发过程的日趋复杂。在早期的嵌入式系统中，软件和硬件是密不可分的，开发人员也大都为电子工程、自动控制领域的工程师，软件也基本上采用汇编语言进行编写。随着软硬件技术的不断发展，人们对嵌入式系统的功能性要求也越来越高，这也就推动了嵌入式系统性能的逐步提高。

与此同时，嵌入式软件的开发和硬件的关系仍旧非常紧密。无论从软硬件系统的平台选择、设计，以至于开发、测试与集成，整个过程都是采取软硬件交互的方式来完成的。因此嵌入式软件的开发已经成为了一项复杂的系统工程，其开发也就必须遵从系统工程和软件工程的要求。

传统的软件生命周期模型是一种顺序的模型，在了解嵌入式软件一般开发流程前，先来看一下传统软件的开发模型。通常自顶向下把软件开发过程分为以下几个阶段：

- 系统定义。
- 需求分析。
- 设计。
- 编码。
- 测试。
- 运行与维护。

在实际的开发过程中，这些阶段是顺序进行的，就像一个瀑布从上到下飞流直下一样，因此也被人们形象地称为瀑布模型。

下面就来简单看一下瀑布模型软件开发过程的几个阶段。

1. 项目的可行性分析

这一阶段的主要工作是对拟开发项目进行可行性论证，主要从以下两个方面入手：

- 技术可行性。
- 资金可行性。

其中技术可行性主要是对现有软件和成熟的技术进行研究，看看能否实现项目的要求；而资金可行性则是看在当前的经济条件下，现有资金是否可以支持该项目的要求。

2. 需求分析

这一阶段需要解决的问题是“做什么”。即根据提出的问题写出需求分析文档，确定要求解的目标、实际的功能、系统的性能以及数据在系统中的流向等，以便为以后的软件开发、设计提供依据。

3. 系统设计

这一阶段要解决的问题是“怎么做”，它分为系统架构设计与详细设计两个阶段。所谓架构

设计就是根据需求分析的结果和系统设计目标等把系统分解成若干个子系统，然后确定系统的总体结构、每个子系统的功能、各个子系统之间是如何关联的等等；而详细设计则是确定每个子系统的名称、每个子系统的功能描述和算法，以及各个子系统的输入/输出信息，各个子系统之间的输入/输出接口等。

4. 编码

这一阶段要在上一阶段的基础上，用某种程序设计语言来编写并调试应用程序。

5. 测试

在实际的程序设计中，一般在需求分析阶段就开始着手测试计划、测试标准、测试方法等内容的编写。测试一般可以分为以下几种：

- 单元测试。
- 集成测试。
- 系统测试。
- 接收测试。

单元测试一般由程序开发人员完成，主要是按照模块或者类进行测试，包含黑盒测试和白盒测试；集成测试是将几个或全部子系统集成后进行的测试，主要测试子系统的接口、关联、功能及所达到的性能，包含黑盒测试和白盒测试；系统测试则是从用户的角度进行测试，包括对最终运行环境的测试、模拟用户的实际操作等，系统测试主要以黑盒测试为主；接收测试则是根据合同和需求分析的内容，测试系统是否已经达到接收的标准。

在这里需要注意的是，上面提到的每一项测试都需要有完成的测试记录，同时开发人员需要根据测试结果来修改程序。

6. 运行与维护

所设计的程序可以良好地运行是软件设计的最终目标，此时设计过程已经基本完成。但是可能会存在前面几个步骤由于考虑不周而导致的小Bug的出现。此时就需要程序开发人员对其进行维护，发现出现问题的原因，并有针对性地加以解决。

瀑布模型具有下面几个优点：

- 为开发项目提供了按阶段划分的检查点。
- 当前一阶段完成以后，我们只需要关注下一个阶段。
- 可以在迭代过程中运用这一模型。

瀑布模型的缺点主要表现在：

- 在项目的各个阶段之间缺少反馈和沟通。
- 只有在软件开发生命周期的最后才可以看到结果。
- 需要通过过多的强制完成日期和里程碑来跟踪各个项目阶段。

为了解决瀑布模型带来的一系列问题，软件开发设计人员又提出了许多软件开发模型，如增量模型、螺旋模型、RUP迭代模型等。但不论哪一种模型，几乎都不能完全适应嵌入式开发的要求。这是因为嵌入式开发中，软件和硬件的关联非常紧密，无论在软件、硬件系统与平台的选择上，还是在软件的设计、开发、测试与集成阶段，都需要软硬件的紧密交互。因此，嵌入式软件的开发应该从整个系统的角度，从全局着眼，需要软件和硬件的协同考虑。

一般情况下，可以把嵌入式软件开发分为下面几个阶段：

1）嵌入式系统的需求获取与需求分析。

2）设计系统的体系结构，选择处理器与相关外部设备，选择操作系统开发平台以及软硬件需求分隔和总体系统集成。

3）软件结构设计、软件的系统设计。

4）软硬件详细设计。

5）软件代码的开发，软件测试与调试。

6）软硬件的联调与集成。

7）系统的测试。

8）系统的运行与维护。

嵌入式系统中软件开发的工作量最多可以占到全部工作量的80%以上，其中测试与调试的工作量占据了很大的工作比重。以我们自主研发的嵌入式平台为例，首先需要明确此嵌入式平台的功能。如可以运行Linux，具有彩色显示、立体声声音播放、可以通过USB接口供电、具有VGA外接显示器功能等等；接下来，就需要考虑嵌入式平台用在哪些行业。因为它应用在工业领域，因此CPU最终选择的是工业级别的CPU。又因为它需要运行Linux以及Windows CE，因此与Windows CE相关的开发工具如Visual Studio 2005、Platform Builder for CE 6.0等就成为了我们的首选；又因为硬件平台的最终运行环境较为恶劣，因此在硬件设计的时候诸如布线等问题就需要重点考虑。在嵌入式平台完成研发以后，就需要进行各个通信接口（如USB接口，并口、串口等）的测试；测试基本无误以后，就需要进行各个接口驱动程序的开发以及操作系统的移植，最后再进行上层应用软件的开发。

从上面的描述中可以看到，嵌入式系统的开发是一个真正的“系统”工程。它不同于传统PC上软件的开发，也不同于单独的硬件设计。实际上，这一独特的开发过程已经很好地体现了嵌入式系统软硬件“融合”这一本质的特征。所谓的“嵌入”在这里已经淋漓尽致地表现了出来。

1.4 嵌入式系统的应用领域

从第一代单片机的出现到今天各种嵌入式设备大规模的应用，嵌入式系统已经具备了将近30年的历史。现在，各种各样的嵌入式设备已经深入到日常生活的方方面面。可以说，我们天天都在和这些嵌入式设备打交道。目前，嵌入式系统主要应用在消费类电子产品、医疗、军事、工业控制、通信和办公等多个领域，下面简单介绍在各个领域中嵌入式设备的使用情况。

1.4.1 在消费类电子产品中的应用

近年来随着技术的不断进步以及竞争的不断加剧，各种消费类电子产品的种类和数量都在急剧地增长。两三年前买一个数码相机需要数千元甚至上万元，而现在1000万像素的数码相机也只需要2500元左右。同时，世界各大家电公司都在不遗余力地推出自己的信息家电和智能家电。据预测，未来的信息家电将会取代计算机成为IT产业新的增长点，各种信息产品也将和网络更为紧密地结合在一起。未来诸如数字电视、机顶盒、数码相机、可视电话和智能玩具等将会走进越来越多的寻常百姓家。

感兴趣的读者也许会问，在我们经常使用的数码相机等消费类电子产品中，哪些地方用到了嵌入式软件及嵌入式技术呢？以数码相机为例，在对摄像过程的控制、数字影像的采集与编辑以

及数码相机和其他电子产品的连接（如直接连接彩色打印机进行数码照片的打印等）等方面都需要嵌入式软件的协助。

尽管嵌入式产品在各类消费类电子产品中都有很重要的应用，但是由于产品研发周期、市场容量等综合因素的影响，嵌入式产品在各种消费类电子产品中有着不同的表现。图1-16展示了嵌入式产品在不同领域中的发展现状及趋势。

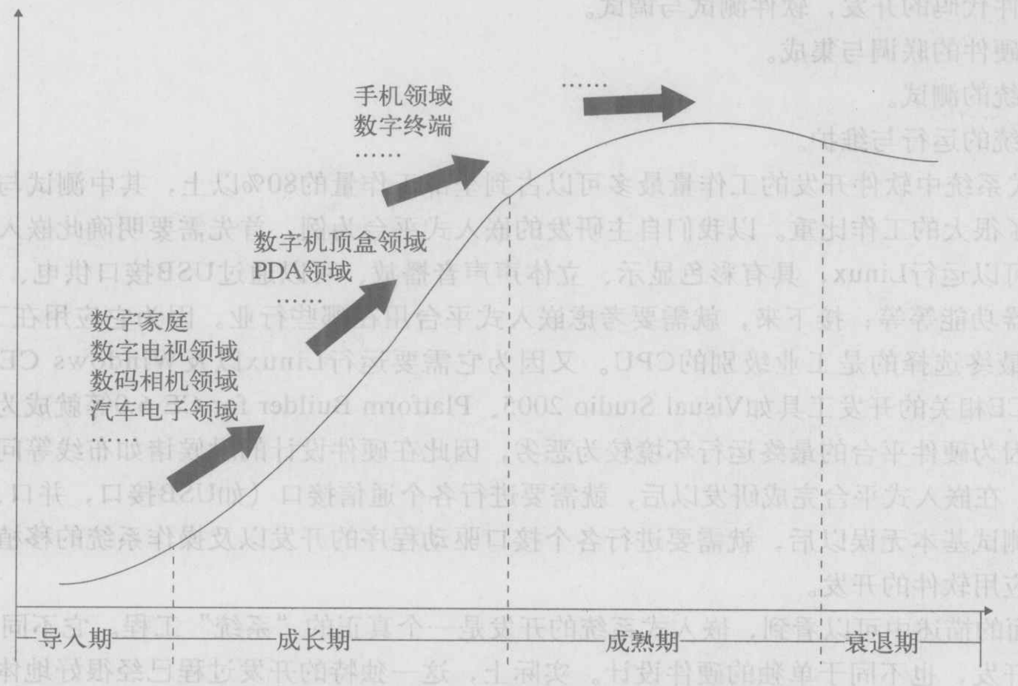


图1-16 嵌入式产品在不同领域中的发展现状及趋势

1.4.2 在工业控制领域的应用

前面曾经提到过，单片机可以被认为是嵌入式行业的开山鼻祖。由于最初的单片机主要应用在工业控制领域，如数控机床、电力设备监控、石油化工系统等，因此工业控制领域就成为了较早应用嵌入式系统的领域。

在目前的工业控制设备中，工控机电产品是使用最多的产品之一。它们一般采用工业级的处理器和各种附加设备来满足实际中对设备较为苛刻的要求。其中采用的处理器以X86架构的嵌入式微处理器居多。在工业控制领域的实际应用中，以前使用较多的16位处理器已经不能满足工业控制领域对设备状态、信息处理速度等方面的要求，因此目前32位、64位的处理器已经逐渐替代了16位的处理器，这样就带动了整个产品性能的提高。

由于ARM系列和MIPS系列等嵌入式微处理器都提供了非常丰富的接口资源，以及方便地实现数据采集、数据处理和通信等功能，因此在工业控制器和设备控制器方面都得到了广泛的应用。

1.4.3 在通信领域的应用

目前嵌入式系统最为广泛的应用就是在通信产品中的应用，其中又以在手机中的应用最为广泛。提到嵌入式芯片在手机中的应用，就不能不涉及手机芯片的两种主要设计方式。一种被称为

单芯片设计方式，另外一种则是双芯片设计方式。

单芯片设计就是把手机中与通信相关的功能以及其他一些辅助功能（如添加多媒体等）都设计在一个芯片上，而与之相对的双芯片设计就是在一个处理器的基础上，再添加一个协处理器。

使用单芯片设计的好处在于：

- 不需要外挂额外的处理器，产品的体积相对较小。
- 使用软件来实现多媒体的应用，灵活性相对更高。
- 只要基带芯片（唯一的单芯片）的功能足够强大，就可以在同样体积大小的手机中不断加入新的功能，如无线功能、TV功能等。
- 制造工艺相对比较简单。

使用单芯片设计的不足之处在于：

- 单芯片需要大容量的存储器来支持其工作，因此对存储器的要求会高一点，功耗相对也会大一点。
- 单芯片的价格一般较高，尤其在功能较多的情况下，因此一般应用在一些高端手机尤其智能手机上。
- 因为需要支持多媒体功能，所以单芯片的设备都需要使用操作系统，因此成本相对较高。
- 因为通信协议和多媒体功能都在一个芯片上运行，所以采用单芯片的手机不论在开发周期上，还是验证周期上都要长一些。
- 越来越快的CPU速度直接导致了功耗的增加，并且在处理某些应用时并不需要那么高的CPU速度，因此会造成资源的浪费。

使用双芯片设计的好处在于：

- 当需要实现多媒体功能的时候，可以通过外挂的多媒体处理器来实现。
- 双芯片的通信平台稳定、省电，而外挂的协处理器由手机的定位决定，可以较快地适应市场的需要。

使用双芯片设计的不足之处在于：因为多媒体功能已经由硬件固定了，当需要加入其他多媒体功能时，就必须使用另外的多媒体处理器。

在高端的智能手机领域，主要有两种不同的体系架构：一种是将通信引擎和应用引擎在同一个芯片上实现的双核架构，即用一个CPU支持两部分的功能；另一种是三核架构，是将通信引擎和应用引擎“分家”，即通信处理器和应用处理器用两个不同的CPU，分别来支持通信功能和应用功能，同时添加一个内核用来负责整个手机系统的控制功能。

随着手机终端用户数量的不断增加，手机等移动设备的销量不断上升，嵌入式设备将会在通信领域得到更大的发展。

1.4.4 在机器人、办公等其他领域的应用

多年以前，我们经常在科幻电影中看到未来机器人会帮助我们做家务，会和我们说话，会协助人类进行危险物品检测等。到了技术快速发展的今天，这些梦想基本上都实现了。机器人技术在机场安检、危险物检查和危险物品排爆等领域得到了广泛的应用。

实际上，机器人技术的发展是和嵌入式技术的发展紧密结合在一起的。嵌入式芯片的发展使得机器人在小型化、智能化方面的优势得以明显的加强。近年来，由于一些32位处理器和

Windows CE.Net等32位嵌入式操作系统的日益流行，使得我们可以通过一个手持式智能设备来实现对机器人的远程信息监控，并且通过无线通信模块控制机器人的运行等功能。例如我们研发的防爆机器人手持控制器就是采用Windows CE 5.0作为操作系统，采用RF无线射频天线作为远距离控制方式的。

在办公领域中使用的复印机、打印机、扫描仪等设备都会用到嵌入式产品；而在商业领域中使用的各类收款机、POS系统，条形码阅读器，商业终端，IC卡设备等也都用到了嵌入式产品和相关技术；在军事领域中，如火炮控制、导弹控制、智能炸弹等武器装备中，也用到了嵌入式产品和技术。

通过上面的介绍，我们会看到，不管是我们日常生活中使用的手机、数码相机、PDA，特定行业领域如煤矿、石油石化、矿山中使用的设备，还是武器装备，如导弹、卫星等都存在着嵌入式的身影。我们有理由相信，随着技术的不断发展和应用领域的不断扩展，嵌入式技术和嵌入式产品必将大行其道，或许有一天，我们的生活方式也将随之而改变。

1.5 嵌入式系统的发展方向

1.5.1 处理器

从前面的介绍中我们知道，嵌入式系统的核心是嵌入式处理器，而嵌入式处理器一般具有下面的特点：

1）对实时多任务有很强的支持能力，能完成多任务并且有较短的中断相应时间，从而使内部的代码和实时内核的执行时间减少到最低限度。

2）具有功能很强的存储区保护功能。这是由于嵌入式系统的软件结构已经模块化，而为了避免在软件模块之间出现错误和交叉作用，需要设计强大的存储区保护功能，同时也有利于软件诊断。

3）可扩展的处理器结构，以便以最快的速度开发出满足应用的高性能的嵌入式微处理器。

4）嵌入式微处理器必须功耗很低。

因此未来嵌入式系统处理器的处理速度会越来越快，功耗会越来越低，可扩展性也会越来越强。例如，基于Windows Embedded CE 6.0的智能手机已经出现了CPU频率为600MHz的产品。图1-17展示了这种手机的外观，图1-18展示了该手机的拨号界面。

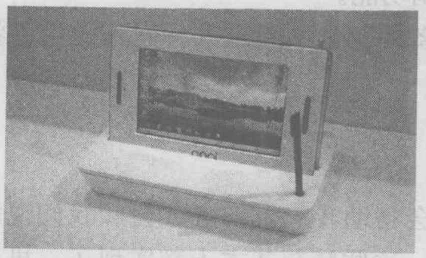


图1-17 采用Windows Embedded CE 6.0系统的智能手机

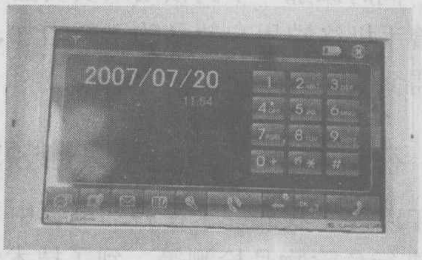


图1-18 手机的拨号界面

1.5.2 图形控制器

早期的PDA等智能设备很少有独立的显卡，因此也就没有独立的显存。但是这种情况正在被改变。著名的显示芯片生产厂商ATI已经发布了用在手机和PDA上的显示芯片。这款芯片具有2D显示引擎、MPEG/JPEG解码器和一个输入/输出装置。显示芯片上的MPEG/JPEG解码器大约占了设备50%的电力，可以提供320×480的图像和视频。微软也打算在下一代PDA上实现多媒体功能，DirectX 8显卡很可能会在下一代PDA上使用，新的移动设备API将会包括诸如Direct3D之类的DirectX 组件。目前一些高端的Windows CE设备已经采用了800×600分辨率的屏幕，对Pocket PC的图形显示和游戏画面功能的提升起到了极大的推动作用。

我们可以预见，在不久的将来，我们的手机、PDA、移动式娱乐终端等都会有独立的显卡，我们可以在上面运行与现在PC上同等效果的游戏、图像处理软件等，那时，智能设备一定会被越来越多的人接受。

1.5.3 智能设备的供电解决方案

再好的设备，没有电的时候也是不能发挥作用的。可以设想一下，当您需要使用手持设备的时候，突然发现没有电了，那个时候你将是一种什么样的心情呢？因此，现在各个公司都在加紧对电池的研究，因为大家都已经意识到电池技术的发展将对PDA等智能设备起着至关重要的作用。尤其是彩屏时代的到来，对大容量电池的需求就更为强烈。

下面就来看看，现在的供电问题是怎样解决的。

在市面上的PDA设备中，用的最多的就是锂离子电池。容量从700mA到1400mA、2000mA不等。锂离子电池的最大特点就是没有记忆效应。这样就可以随用随充电，不像原来的镍氢和镍镉电池有很强的记忆效应，在充电之前要把电池的电量放光。尽管锂离子电池有这么大优势，但是还有不足之处。首先就是环境的破坏。尽管和镍氢、镍镉电池相比，对环境的影响没有它们那么大，但是离真正的无害化还有一段距离。要知道，现在各国政府对环境的问题都是非常重视的，如果对环境的破坏问题解决不了，有一天就会面临被禁止销售的危险。其次就是锂离子电池的价格比较昂贵，尤其是原装的电池。现在世界上锂离子电池最大的生产商是日本的三洋公司，约占全球锂离子电池的70%多。另外一家大厂就是我们熟悉的日本SONY公司，产量约占世界的20%。不要小看这小小的电池，目前高端的智能锂离子电池我们国家还是不能生产的。核心的技术都是掌握在日本厂家的手里。所谓的智能型锂离子电池，就是指这种电池与相应的硬件和软件结合在一起，就会显示出电池的电量、剩余使用时间、电池的损耗程度等信息。例如，SONY DSC P1数码相机就可以显示剩余的使用时间，精确到分钟。尼康公司新推出的D2H数码相机使用的EN－EL4型智能锂离子电池就可以显示电池的裂化程度，也就是电池的损耗程度。

除了常用的锂离子电池之外，现在还出现了燃料电池、太阳能电池、生物电池等新型电池。这里面提到的燃料电池是一种直接将储存在燃料和氧化剂中的化学能，等温、高效（50%～70%）、环境友好地转化为电能的发电装置。它具有能量转换效率高、无污染、噪声低、适应不同功率要求等优点，被认为是21世纪全新的高效、节能、环境友好的发电方式之一。由于燃料电池是以甲醇为主要的原材料，因此使用后的产物主要是水。生成的水可以当作稀释剂，加入高浓度的甲醇后浓度在3%～6%之内即可再次使用了。不但体现了环保精神，也利于用户节约资金，降低了用户的使用成本。

可以说，供电技术的发展已经成为了嵌入式设备发展的一个瓶颈问题。同时随着嵌入式设备中更快速度的处理器、更高分辨率的屏幕、更强大的多媒体应用的出现，供电时间和性能之间的矛盾变得更为突出。只有解决好这个问题，嵌入式设备的发展才能取得长足的进步。

1.5.4 网络技术的应用

之所以把这一部分单独拿出来进行介绍，是因为网络技术的大规模运用对嵌入式设备的发展起到了巨大的推动作用。如通过实时地接入互联网，我们可以把嵌入式设备远程监控或输入的数据实时传送到后方的数据分析中心，使得后方指挥更加及时和准确。

图1-19展示了在各地政府中广泛应用的“城管通”手持设备，图1-20展示了该种手持设备的软件界面。

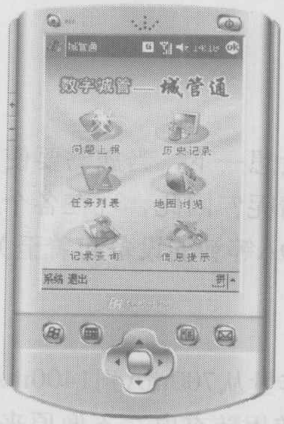


图1-19 “城管通”设备

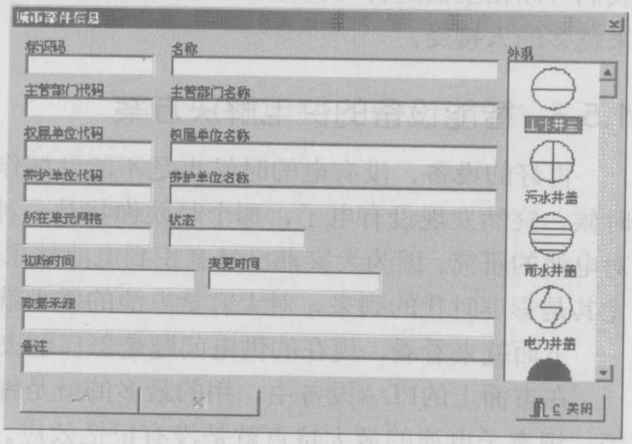


图1-20 “城管通”软件界面

在这里需要特别提及的是，网络技术的应用不但需要嵌入式设备硬件的支持，如网卡等，还需要软件的协同运作，如浏览器等，才可以实现网络功能。

最直接的上网方式就是用Windows Embedded CE 6.0设备自带的ActiceSync软件。当Windows CE设备和能够上网的计算机连接在一起以后，在设备中直接运行IE浏览器，然后输入网址，就可以通过计算机上网了。这种方式是最直接的一种方法。但是也存在一定的问题。那就是当身边的计算机可以上网的时候，有几个人会用Windows CE设备上网呢？毕竟嵌入式设备和PC的性能还是不能同日而语的。

这几年无线上网兴起，作为时尚的嵌入式设备用户自然不会错过这个时机。

无线上网的方式之一就是通过Windows CE等嵌入式设备的红外线端口和支持拨号上网的手机相连接，然后通过手机拨号上网。当然手机也要支持红外线端口。这种方式的特点就是可以实现移动上网。如当您出差在外地的时候，有紧急公务需要处理，而身边又没有可以上网的计算机，网吧的保密性又太差，这时只要您身边有一台PDA和一部支持拨号上网的手机就可以实现您的办公梦想了。但是由于红外线传输的特点，在上网的时候，PDA的红外线端口和手机的红外线端口必须直对（有一个小的角度也可以），因此不是很方便。

无线上网的第二种方式就是通过蓝牙和手机连接在一起，通过手机拨号上网。现在的高端PDA都内置蓝牙功能，而高端的手机也支持蓝牙功能。这样当两者通过蓝牙连接在一起的时候，就可以实现随时随地办公了。蓝牙的特点在前面已经介绍过，就是不受方向角度的限制，只要在

有限距离内，就可以连接成功。

第三种方式就是现在最流行的通过802.11a/b/g协议进行无线上网。需要的条件就是一个接入点（Access Port），一个支持802.11a/b/g协议的PDA。这种方式和前两种相比最大的优点就是距离大大的增加了，并且速度可以达到局域网的速度，这个速度可不是那种拨号上网可以相比的。

既然提到了802.11a/b/g协议，那么就在这里对这些协议做一个简单的介绍。

（1）IEEE 802.11a

1999年，IEEE 802.11a标准制定完成，该标准规定无线局域网工作频段在5.15～5.825GHz，数据传输速率达到54/72Mbit/s(Turbo)，传输距离控制在10～100m。802.11a采用正交频分复用（OFDM）的独特扩频技术；可提供25Mbit/s的无线ATM接口和10Mbit/s的以太网无线帧结构接口，以及TDD/TDMA的空中接口；支持语音、数据、图像业务；一个扇区可接入多个用户，每个用户可带多个用户终端。

（2）IEEE 802.11b

1999年9月IEEE 802.11b被正式批准，该标准规定无线局域网工作频段在2.4～2.4835GHz，数据传输速率达到11Mbit/s。该标准是对IEEE 802.11的一个补充，采用点对点模式和基本模式两种运作模式，在数据传输速率方面可以根据实际情况在11Mbit/s、5.5Mbit/s、2Mbit/s、1Mbit/s的不同速率间自动切换，而且在2Mbit/s、1Mbit/s速率时与802.11a兼容。802.11b使用直接序列（Direct Sequence）DSSS作为协议。802.11b和工作在5GHz频率上的802.11a标准不兼容。由于价格低廉，802.11b产品已经被广泛地投入市场，并在许多实际工作场所运行。

（3）IEEE 802.11g

2003年6月，IEEE推出最新版本IEEE 802.11g认证标准，该标准提出拥有IEEE 802.11a的传输速率，安全性较IEEE 802.11b好，采用两种调制方式，含802.11a中采用的OFDM与IEEE 802.11b中采用的CCK，做到与802.11a和802.11b兼容。虽然802.11a较适用于企业，但无线局域网运营商为了兼顾现有802.11b设备投资，选用802.11g的可能性极大。

采用这种方式上网也存在一些问题。最重要的一个就是安全问题。因为是无线上网，因此传输的数据报很容易被中途监听，这样就会对用户产生安全的隐患。尤其是作为军事、科研、商业的用户，对安全的问题更为敏感。尽管IEEE 802.11g 的安全性已经有了很大的提高，但是和人们的期望相比还有一段的距离。

第四种方式就是使用Windows CE设备通过GPRS或者CDMA网络连接到互联网。这种嵌入式设备一般都具备手机功能，不但可以实现嵌入式设备的一个基本功能，还可以进行一系列与手机相同的操作。目前，这种上网方式已经成为很多智能手机的标准功能，真正实现了随时随地接入互联网。

1.6　几种常见的嵌入式操作系统

同传统的PC一样，嵌入式系统的功能日益庞大，结构日益复杂。传统的前后台控制程序已经不能很好地满足系统性能的要求了。因此在这里引入了嵌入式操作系统，以实现对软件资源和硬件资源合理、高效的管理。目前在世界范围内有很多的嵌入式操作系统设计厂商，据统计全世界约有200多种嵌入式操作系统可供嵌入式系统设计人员选择。常见的嵌入式操作系统主要有Windows CE、Linux、VxWorks、Palm和Symbian等。下面首先介绍嵌入式操作系统和传统操作

系统相比，具有哪些特点。

1.6.1 嵌入式操作系统的特点

我们知道，一台计算机系统里面的操作系统具有非常重要的作用。设备所运行的软件，对外接设备的识别和操作，以及对用户指令的响应都离不开操作系统。在传统PC领域，操作系统的作用主要体现在：

1）进程管理。

2）文件管理。

3）存储管理。

4）设备管理。

5）作业管理。

其中进程管理是指对处理器资源的管理，对各种进程状态进行控制；文件管理是指对外存设备的管理，如计算机的程序和所用数据以什么方式存储在外部存储设备中，如何对其进行读写操作等；存储管理则主要是指如何对内存资源进行管理；设备管理主要是指对输入/输出设备的操作；作业管理则体现在对计算机用户操作的管理上。需要特别指出的是，尽管操作系统中一般都会包含上述功能，但是不同的操作系统其功能实现的方式各不相同，这也就导致了不同的操作系统性能上有很大的差异。

那么和传统PC领域的操作系统相比，嵌入式操作系统又具有哪些不同呢？下面就简单介绍一下。

1）设计原则不同。传统PC领域中的操作系统主要注重系统的整体性能，同时强调对多用户多任务的支持及系统的平均响应时间，这一点从我们使用的Windows XP等系统上就可以体会到。而嵌入式系统则大多针对“专一”的某种功能和强调很高的实时性，如城市管理部门用到的“城管通”系统就会专注于城市管理相关的内容（井盖的位置，井盖是否丢失，街道路面是否有损坏等），而类似于媒体播放这样的功能就显得不是很重要。

2）交互方式不同。传统PC领域中的操作系统同用户交互模式一般体现在鼠标、键盘、摄像头等外接设备上，而嵌入式操作系统除了一些外接设备以外，还可以通过触摸屏、电子墨水笔、系统自带的语音识别软件等实现人机交互。

3）稳定性要求不同。PC领域操作系统的稳定性已经越来越高，如Windows XP的稳定性已经大大优于原来的Windows 98，但是和嵌入式相比仍旧存在很大的差距。如应用在航空航天领域和工业控制领域的嵌入式设备，如果没有高的稳定性和可靠性是无法适应其使用环境的。

4）开发环境不同。传统PC领域开发出来的软件可以在不同PC上运行，而不需要过多地考虑底层硬件设置。而嵌入式领域的软件设计则需要直接对底层硬件进行操作才可以获得良好的软件性能。

5）系统资源不同。PC上的操作系统可以利用PC硬件资源非常丰富的特点充分发挥软件性能，而嵌入式系统的资源大多有限（ROM空间有限，RAM资源不足，CPU频率不高），因此要想实现软件的高性能，就得在算法设计等方面做足文章。

6）实时性的要求不同。嵌入式设备大多用在对实时性要求很高的领域，尤其是一个工业控制领域的产品，因此对实时性相应的要求更高。

7）可裁剪性不同。嵌入式系统可以根据实际应用来进行内核的裁剪，这一点PC领域的操作系统很难实现。

那么针对这些特点，嵌入式操作系统又分成哪些种类呢？下面就简单介绍一下。

1.6.2 嵌入式操作系统的分类

目前的嵌入式操作系统按照应用范围可以分为通用型嵌入式操作系统和专用型嵌入式操作系统。比较典型的通用型操作系统有Windows CE、Linux和VxWorks等。专用型操作系统有Smart Phone、Pocket PC和用于移动电话的Symbian等。通用型操作系统在应用到实际的环境中时，一般都要经过重新的定制，以适应具体环境的要求；而专用型操作系统在应用的时候可以不经过裁减直接应用，或者只需要很少的设置就可以使用。

如果按照实时性来划分，嵌入式操作系统又可以分为以下两类。

1. 实时嵌入式操作系统

它主要针对通信和控制领域，如VxWorks等。其中实时系统又分为硬实时系统和软实时系统。

- 硬实时系统：如果系统在指定的时间内没有完成某项特定的任务或没有完成某个特定的操作，那么将会导致系统的验证错误，甚至导致系统的整体崩溃。在实现上，硬实时系统通常情况下会在硬件上添加一个专门用于实践管理和优先级管理的控制芯片。
- 软实时系统：在这种系统中，迅速地作出对系统的响应同样重要。与硬实时系统所不同的是，当在设定的时间内没有完成任务或者指定操作的时候，不会导致系统致命的错误。在实现上，软实时系统主要通过软件编程来实现对时间的管理。

2. 非实时嵌入式操作系统

这种嵌入式操作系统主要应用在消费类电子领域，包括掌上电脑、手机、机顶盒等。微软推出的面向手机应用的Smart Phone就属于这种类型。

1.6.3 Windows Embedded CE

因为本书中所讲述的内容都是以Windows CE为平台进行讲解的，因此首先介绍一下Windows CE系统的特点。

Windows CE系统是微软公司推出的一种32位的嵌入式操作系统。它具有很好的模块性、良好的多媒体功能和友好的人机交互界面。同时，Windows CE还很好地支持了PowerPC、X86、ARM、MIPS等多个系列处理器的体系结构，并为所支持的处理器提供了非常完善的系统库。随着技术的不断发展，以及市场的需要，微软公司在Windows CE中也添加了桌面PC系统上使用的DirectX技术。为了更好地保证网络环境下设备的良好运转，在Windows CE中也添加了对广域网、多媒体、LAN接口标准、蓝牙、红外和无限网络传输等协议的支持。

Windows CE系统一般主要包含4个模块。它们分别为：

- 内核模块。
- 对象存储模块。
- 图像、窗口和事件子系统模块。
- 通信模块。

和其他常见的操作系统一样，内核是Windows CE的核心。它主要负责内存管理、进程间调

度等功能。同时还要管理虚拟存储、多任务处理等。通信模块则主要负责对I/O接口的支持、远程访问服务、局域网、无线服务和相应的传输控制协议等。

在这里提到的进程是指程序的一次动态执行过程，也是操作系统分配系统资源的最基本的单位。在Windows CE中，系统最多支持32个进程，同时系统启动的时候至少要创建4个进程。而线程是操作系统调度的最基本的单位。线程本身不拥有系统资源，但是属于一个进程的线程间共享该进程所拥有的全部资源。进程创建线程，线程是进程的一个实体。

图1-21展示了Windows CE的体系结构。

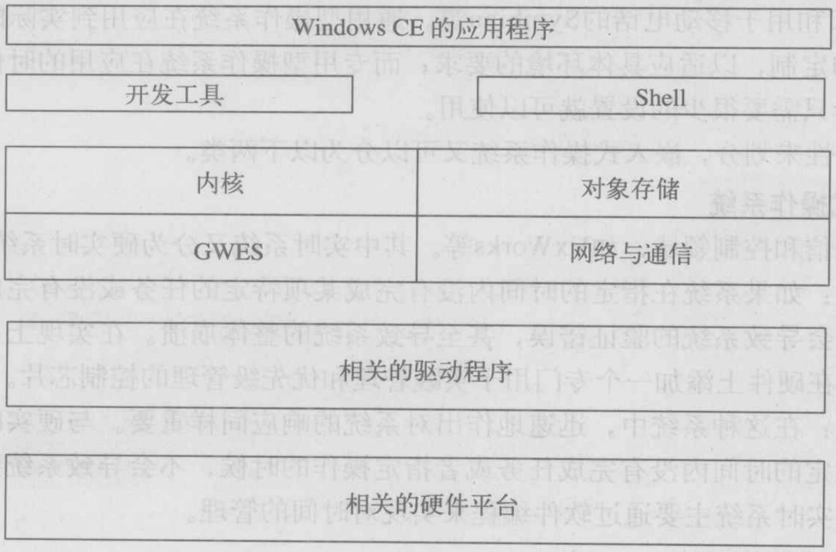


图1-21　Windows CE的体系结构

图1-22展示了采用Windows Mobile 5.0的智能手机，图1-23展示了在Windows Mobile 5.0上运行游戏程序时的画面。

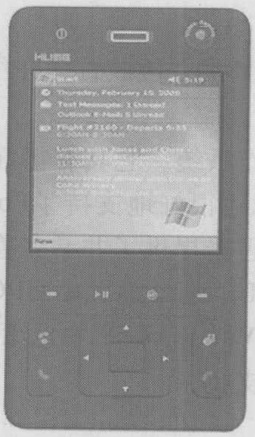


图1-22　基于Mobile 5.0的智能手机

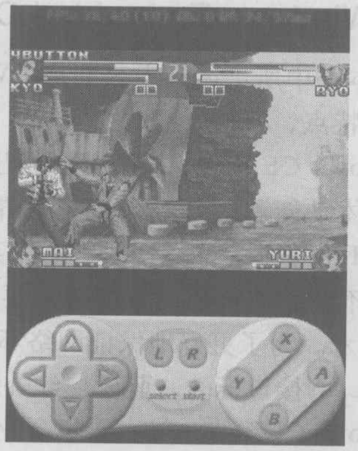


图1-23　在Mobile 5.0上运行的游戏程序

1.6.4 Symbian

Symbian是由摩托罗拉、西门子、诺基亚等几家大型移动通信设备商共同出资组建的一个合资公司（摩托罗拉已经撤资），专门研发手机操作系统。2008年6月，诺基亚成为Symbian的独资

公司。Symbian操作系统的前身是EPOC，而EPOC是Electronic Piece of Cheese取第一个字母而来的，其原意为使用电子产品时可以像吃乳酪一样简单，这就是它在设计时所坚持的理念。Symbian不仅仅是一个操作系统的名称，而且还是一个公司、一个阵营、一个手机垄断联盟的代名词。不同公司根据自身利益的需要对这一阵营也采取了不同的态度。例如，摩托罗拉曾在2004年就卖掉了所持Symbian股份。

Symbian开放式的操作系统中，有两种类型，一类用于基于掌上电脑、带手写笔的智能手机（专区），代表产品有索尼爱立信的P802等；另一类则用于HPC手机，代表产品有诺基亚9210等。由于该操作系统出产于由摩托罗拉、西门子、诺基亚等几家大型移动通信设备商共同出资组建的一个合资公司，所以，它常用于上述厂商推出的相应的手机上。例如我们熟悉的索尼爱立信的P802和P908、诺基亚7610、西门子SX1等所用的就是该操作系统。

图1-24展示了P908手机，图1-25展示了Nokia E90手机。

图1-24 P908手机

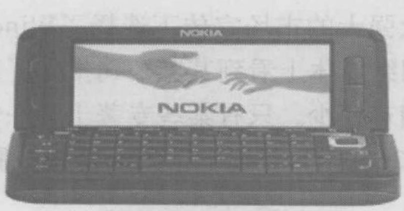


图1-25 E90手机

就目前来说，Symbian主要应用在手机设备上，尤其是高端的智能手机上。借助诺基亚作为全球第一大手机制造商的优势，在目前的高端智能手机领域，Symbian已经占据了世界第一的位置。

与其他移动终端操作系统相比，Symbian具有以下两个特点：

1）由于Symbian起初为手机专用操作系统，因此其对移动通信协议如GSM、GPRS、WCDMA、蓝牙以及CDMA 2001X等的支持优于其他系统。在手机用户的接受程度、手机软件的易用性、运营商的合作等方面都有一定优势。

2）Symbian由众多顶级传统手机厂商联合成立，公司的股东诺基亚和索尼爱立信等几家手机巨头的支持也是其他竞争对手不可企及的巨大优势。凭借这些手机厂商生产上的技术优势和市场经验，其智能手机在未来若干年的国际市场中都会占有很大份额。但同时，这些手机厂商之间存在既竞争又合作的关系，这也是限制Symbian发展的致命弱点。企业内部存在着许多诸如专利、费用、地位等问题，这些势必导致各公司之间的分歧，影响技术发展。

尽管目前Symbian已经占据了高端手机市场世界第一的位置，但是它也正面临着来自Windows Mobile和Linux的强力挑战，内忧外患给Symbian带来了巨大的压力。但无论如何，具有多年的发展经验、拥有众多顶级厂商的支持以及广泛的用户认识度的Symbian仍将在未来几年内保持优势地位。

1.6.5 Palm

Palm是3COM公司出品的一种嵌入式操作系统。它是专门为手持计算领域所设计的一个系统。

在早期，Palm得到了IBM、Nokia、SONY等众多国际知名厂商的支持，也一度占据了市场第一的份额。但是随着微软Windows CE以及Pocket PC、Smart Phone等嵌入式操作系统大举进军移动计算领域，Palm的市场份额也在逐年下滑。SONY公司前些年宣布退出PDA领域更是为Palm阵营雪上加霜。尽管如此，目前还有很多人热衷于Palm操作系统，并且这些人被亲切地称为“胖友”，而与之相对的喜欢Windows CE的人则被称之为“蜥蜴”。就目前的发展态势来看，Windows CE大有继续扩大市场份额、继续攻城掠地的势头。

Palm系统提供了一个基本任务管理的抢占式多任务管理内核。在存储器管理中也引入了虚拟文件系统API接口来管理相当于外存的部分存储器。在Palm中，主要是由用户接口程序外壳负责管理正在运行的用户接口程序。和Windows CE不一样，Palm中的内存和外存都集中存储在随机存储器中。

Palm操作系统一个最大的特点就是节省电能以及占用很少的系统资源。同时，它还采用开放式的架构，因此吸引了众多为Palm平台开发应用软件的程序设计人员。据统计，目前可以在Palm平台上运行的共享软件与免费软件已经超过一万种，各种优良的应用软件还在不断的扩增之中。

在众多国内的用户当中，喜爱Windows CE的人数正在逐年的扩展，尤其是年轻人，很多人都在微软强大的市场宣传下选择了Windows CE。如果经常看IT方面的信息，一定会从报刊、杂志以及网络媒体上看到某某公司推出了基于Mobile 5.0的Pocket PC等信息。相比较之下，Palm的信息则相对较少。只有某些专著于这一领域的专业人士和发烧友才会更为关心Palm的发展。

图1-26和图1-27列出了Palm掌上电脑和Palm操作系统的界面。

图1-26 Palm掌上电脑

图1-27 Palm操作系统的界面

不管是Windows CE还是Palm，都会面对市场竞争带来的压力。一方市场份额的不断扩大，毕竟会带来另一方市场份额的下降。各大厂家为了自身利益而“化干戈为玉帛”的事件也时有发生。昔日PDA市场的贵族Palm为了自身利益也不得不与昔日的对头微软进行联手。

作为PDA市场的先驱，Palm曾风光一时，仅次于业内的“一号人物”Symbian，遥遥领先于微软。在前面已经提到，对于Palm类产品，人们昵称为“胖友”，而对于Windows CE类产品，则被称之为“蜥蜴”。而在鼎盛时期，Palm曾占有高达90%以上的市场比例。这一记录，微软恐怕只能在桌面操作系统可以创造。

但是，在过去的几年里，由于受到微软和生产“黑莓”无线电子邮件设备的RIM公司的激烈竞争，Palm形势急转直下，市场份额也一路下滑。在2004年第2季度，全球销售的PDA产品中41.8%都采用了Palm操作系统，而采用微软视窗操作系统的比例为36.6%。然而，到了2005年第2季度，Palm的领先地位消失殆尽，采用Palm操作系统的PDA比例仅为18.8%，而微软的比例上升到45.7%。

同时，由于出现一系列战略失误，Palm渐渐被竞争对手超越。Palm操作系统只注重最基本的功能，而忽视了用户对于多媒体以及网络功能的需求。Palm一方面出售PDA产品，另一方面又试图通过向其他公司许可其PDA软件系统来构筑其软件业务，而这最终会造成硬件与软件业务之间的“自相矛盾”式冲突。

为应对此种尴尬局面，2003年，Palm一分为二，分别成立了Palm one（硬件）和Palm source（软件）两大独立公司。Palm希望通过分拆能向更多的厂商出售Palm OS，但事与愿违，在索尼宣布退出PDA市场以后，Palm source的新客户廖廖无几，而具有血缘关系的Palm one成为唯一的客户。2005年9月，日本的Access公司以3.24亿美元的价格收购了Palm source。在这种情况下，Palm one不得不回购Palm商标使用权，并重新把公司命名为Palm。

实际上，Palm和微软走到一起，既是面对残酷的市场现实，也是一种无奈之举，更是一招绝地反击的险棋。Palm的CEO埃德·科林根（Ed Colligan）曾经感叹地承认，两个长期争锋相对的竞争对手如今走到了一起，其中不无讽刺性。他表示：“如果你生活经验足够丰富，你会发现，合作伙伴有时会是竞争对手，而竞争对手有时也会成为合作伙伴。”

可以预料，与微软合作可能为Palm打开一个巨大的企业用户市场，原因是微软在台式机软件市场上处于绝对垄断地位，从而使其成为企业一个不可或缺的供应商。通过合作，Palm智能手机绝地反击，也许会因此重获新生。但是，Palm OS走向没落将是一个必然的趋势。尽管Palm依然对Palm OS恋恋不舍，表示将推出同时支持两个操作系统的产品。但是，现委身日本Access公司的Palm source已是身不由己，其很可能会中断Palm OS系统的开发，即使这一系统继续存活，它也将风光不再，沦为配角，会被应用到一些低端手机和PDA产品中。

业内认为，Palm未来会同时出售Palm OS和Windows Treos，随着时间的推移，微软的操作系统肯定会占据市场的主导地位。

图1-28展示了Palm推出的第一款基于Windows Mobile智能手机Treo 700w；图1-29展示了Treo 700w的图形用户界面。

图1-28　Treo 700w智能手机

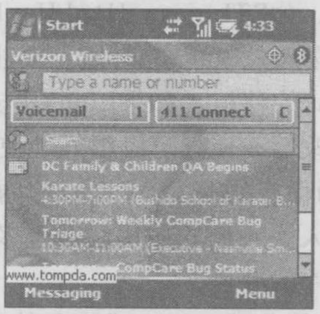


图1-29　Treo 700w智能手机界面

1.6.6　嵌入式Linux

在1991年的时候，互联网还不像现在这么普遍，上网的人大部分是一些研究机构或者是大学里面的学生、教授。其中有一个叫Linus Torvalds的年轻芬兰大学生在comp.os.minix这个新闻群组上发表了这样一个帖子：

Hello everybody out there using minix-

I m doing a (free) operation system (just a hobby,

won't be big and professional like gnu) for 386(486) AT clones.

这个Linus的兴趣就是Linux的核心0.02版的原始代码，当初还是个大学生的 Linus大概完全没想到当初由于他个人爱好而出现的Linux会有成千上万个使用者，而由他自己带头开发的操作系统现在已经在世界各地受到普遍的欢迎。

准确地说，Linux是一套类UNIX的操作系统，也属于UNIX的一种。Linux最为吸引人的地方就是使用者在网络上就可以看到Linux的原始代码。在互联网日渐盛行以及 Linux开源魅力的吸引下，无数的电脑高手参与到其中，进行开发、改善 Linux的核心代码，使得Linux的功能日见强大。

在开源思想的带动下，Linux上的软件越来越丰富。我们几乎可以在任何一个领域中找到Linux下的软件。同时Linux的内核非常稳定，由于它的稳定性，Linux已经在各个领域，尤其是在网络服务器领域得到了非常广泛的应用。它的内核小巧灵活，易于裁减，这使得它更加适合于嵌入式系统的应用。目前Linux已经可以很好地支持多种体系结构，如X86、MIPS、ARM等。作为系统中非常重要的部分，设备驱动程序在Linux中也已经得到了很好的发展和应用。目前Linux上的设备驱动程序已经非常的完善和丰富，同时支持各种主流硬件设备和最新的硬件技术。我们知道，如果进行Windows下的程序开发工作，很多开发工具（如Visual Studio、Platform Builder等）的价格是非常昂贵的。但是在Linux下，很多开发工具都是免费的，并且从高层应用到底层开发，Linux下都已经具备了非常完善的开发工具链。

作为Linux的核心，Linux的内核同时使用了C语言和汇编语言来实现。使用这两种语言的优势在于，C语言编写的代码移植性较好，易于维护，而汇编语言编写的程序则速度较快。一般只有在速度是关键因素或者一些因平台相关特性而产生的特殊要求（例如直接和内存管理硬件进行通信）时才采用汇编语言。

作为应用在嵌入式领域的嵌入式Linux，主要可以分为两类：第一类是在利用原有Linux强大功能的前提下，简化Linux的某些功能，使它的规模尽可能地小，以满足嵌入式系统对体积的要求，如uClinux等；第二类是将Linux开发成实时系统，尤其是硬实时系统，主要应用在一些关键的控制场合，如RTLinux、Hard Hat Linux等。

在开发嵌入式Linux的时候，要注意以下几个问题：

1）Linux的移植。如果Linux不支持选用的平台，那么就需要把Linux内核中与硬件平台相关的部分进行改写，使之可以很好地使用选用的平台。

2）驱动程序的开发。Linux的内核更新的很快，因此许多最新的硬件驱动很快就可以得到支持。但是嵌入式系统的应用领域是多种多样的，所选用的硬件设备也各不相同，因此要格外重视设备驱动程序的开发。

3）内核的裁减。嵌入式产品的可用资源比较少，所以Linux的内核相对嵌入式系统来说就显得有点大，需要对其进行裁减，以达到可以利用的程度。

4）对中文字体的支持。

5）应用软件的开发与支持。由于Linux上的软件数量远远不如Windows下的软件，因此在开发的时候就会遇到一定的困难。

图1-30为新款Linux手机，图1-31为新款Linux智能手机操作界面。

目前Linux主要应用在信息家电、PDA、机顶盒、数据网络、远程通信、医疗、电子交通运输、各种计算机外设、工业控制、航空、航天领域等。

图1-30 Linux智能手机

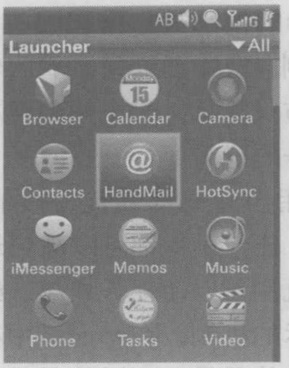


图1-31 新款Linux智能手机操作界面

1.6.7 VxWorks

VxWorks操作系统是由美国WindRiver公司于1983年设计开发的一种嵌入式实时操作系统（RTOS），是嵌入式开发环境的关键组成部分。具有良好的持续发展能力、高性能的内核以及友好的用户开发环境，VxWorks在嵌入式实时操作系统领域占据一席之地。它以其良好的可靠性和卓越的实时性，广泛地应用在通信、军事、航空、航天等高精尖技术以及实时性要求极高的领域中，如卫星通信、军事演习、弹道制导、飞机导航等。在美国的F-16和FA-18战斗机、B-2隐形轰炸机和爱国者导弹上，甚至连1997年4月在火星表面登陆的火星探测器上也使用了VxWorks。

1. VxWorks的主要特点

1）非常高的可靠性。

操作系统的用户希望在一个工作稳定、可以信赖的环境中工作，所以操作系统的可靠性是用户首先要考虑的问题。而稳定、可靠一直是VxWorks的一个突出优点。自从对中国的销售解禁以来，VxWorks以其良好的可靠性在中国赢得了越来越多的用户。

2）良好的实时性。

实时性是指能够在限定时间内执行完规定的功能并对外部的异步事件作出响应的能力。实时性的强弱是以完成规定功能和作出响应时间的长短来衡量的。由于VxWorks系统本身的开销很小，进程调度、进程间通信、中断处理等系统公用程序精练而有效，因此它们的延迟很短。VxWorks提供的多任务机制中对任务的控制采用了优先级抢占（Preemptive Priority Scheduling）和轮转调度（Round-Robin Scheduling）机制，也充分保证了可靠的实时性，使同样的硬件配置能满足更强的实时性要求，为应用的开发留下更大的余地。

3）良好的可裁减性。

用户在使用操作系统时，并不是操作系统中的每一个部件都要用到。例如图形显示、文件系统以及一些设备驱动在某些嵌入系统中往往并不使用，因此VxWorks由一个体积很小的内核及一些可以根据需要进行定制的系统模块组成。VxWorks内核最小为 8kB，即便加上其他必要模块，所占用的空间也很小，且不失其实时、多任务的系统特征。由于它的高度灵活性，用户可以很容易地对这一操作系统进行定制或作适当开发，来满足自己的实际应用需要。

2. VxWorks的主要组成部分

（1）高性能的实时操作系统核心wind

VxWorks的核心被称作wind，包括多任务调度（采用优先级抢占方式），任务间的同步和进程间通信机制以及中断处理，看门狗和内存管理机制。一个多任务环境允许实时应用程序以一套独立任务的方式构筑，每个任务拥有独立的执行线程和它自己的一套系统资源。进程间通信机制使得这些任务的行为同步、协调。

wind使用中断驱动和优先级的方式。它缩短了上下文转换的时间开销和中断的时延。在VxWorks中，任何例程都可以被启动为一个单独的任务，拥有它自己的上下文和堆栈。还有一些其他的任务机制可以使任务挂起、继续、删除、延时或改变优先级。wind核提供信号量作为任务间同步和互斥的机制。在wind核中有几种类型的信号量，它们分别针对不同的应用需求：二进制信号量、计数信号量、互斥信号量和POSIX信号量。所有的这些信号量是快速和高效的，它们除了被应用在开发设计过程中外，还被广泛地应用在VxWorks高层应用系统中。对于进程间通信，wind核也提供了诸如消息队列、管道、套接字和信号等机制。

（2）I/O系统

VxWorks提供了一个快速灵活的与ANSIC兼容的I/O系统，包括UNIX标准的缓冲I/O和POSIX标准的异步I/O。VxWorks包括以下驱动程序：网络驱动、管道驱动、RAM盘驱动、SCSI驱动、键盘驱动、显示驱动、磁盘驱动、并口驱动等。

（3）文件系统

VxWorks提供的快速文件系统适合于实时系统应用。它包括几种支持使用块设备（如磁盘）的本地文件系统。这些设备都使用一个标准的接口从而使得文件系统能够被灵活地移植在设备驱动程序上。另外，VxWorks也支持SCSI磁带设备的本地文件系统。VxWorks I/O体系结构甚至还支持在一个单独的VxWorks系统上同时并存几个不同的文件系统。VxWorks支持四种文件系统：dosFs、rt11Fs、rawFs和tapeFs。

另一方面，普通数据文件、外部设备都统一作为文件处理。它们在用户面前有相同的语法定义，使用相同的保护机制。这样既简化了系统设计又便于用户使用。

（4）板级支持包BSP（Board Support Package）

板级支持包对各种板子的硬件功能提供了统一的软件接口，它包括硬件初始化、中断的产生和处理、硬件时钟和计时器管理、局域和总线内存地址映射、内存分配等。每个板级支持包包括一个ROM启动（BootROM）或其他启动机制。

（5）网络设备

VxWorks提供了对其他网络和TCP/IP网络系统的“透明”访问，包括与BSD套接字兼容的编程接口、远程过程调用（RPC）、SNMP（可选项）、远程文件访问（包括客户端和服务端的NFS机制以及使用RSH、FTP或TFTP的非NFS机制）以及BOOTP和ARP代理。无论是松耦合的串行线路、标准的以太网连接还是紧耦合的利用共享内存的背板总线，所有的VxWorks网络机制都遵循标准的Internet协议。

（6）先进的网络产品

VxWorks内的WindNet是先进的、系列的网络产品，这些产品扩展了VxWorks的网络特性，并增强了嵌入式处理器的网络特性。

（7）虚拟内存（即VxVMI选项）与共享内存（即VxMP选项）

VxVMI为带有MMU的目标板提供了虚拟内存机制。VxMP提供了共享信号量，消息队列和在

不同处理器之间的共享内存区域。

(8) 目标代理（Target Agent）

目标代理遵循WBD（WindDebug）协议，允许目标机与主机上的Tornado开发工具相连。在目标代理的默认设置中，目标代理是以VxWorks的一个任务——tWdbTask的形式运行的。Tornado目标服务器（TargetServer）向目标代理发送调试请求。调试请求通常决定目标代理对系统中其他任务的控制和处理。默认状态下，目标服务器与目标代理通过网络进行通信，但是用户也可以改变通信方式。

(9) 实用库

VxWorks提供了一个实用例程的扩展集，包括中断处理、看门狗计时器、消息登录、内存分配、字符扫描、线缓冲和环缓冲管理、链表管理和ANSIC标准。

(10) 基于目标机的工具

在Tornado开发系统中，开发工具是驻留在主机上的。但是也可以根据需要将基于目标机的Shell和装载卸载模块加入VxWorks。

通常情况下，仅仅依靠人工编程调试很难发挥它的功能并设计出可靠、高效的嵌入式系统，必须要有与之相适应的开发工具。TornadoII就是为开发VxWorks应用系统提供的集成开发环境，TornadoII中包含的工程管理软件，可以将用户自己的代码与VxWorks的核心（wind）有效地组合起来，可以按用户的需要裁减配置VxWorks内核；VxSim原型仿真器可以让程序员在不用目标机的情况下，直接开发系统原型，作出系统评估；功能强大的CrossWind调试器可以提供任务级和系统级的调试模式，可以进行多目标机的联调；优化分析工具可以帮助程序员从多种方式真正地观察、跟踪系统的运行，排除错误，优化性能。

3. VxWorks的主要应用领域

- 消费类电子产品，如掌上电脑、机顶盒、可视电话、汽车导航系统等。
- 航空航天领域，如飞行模拟器、航班管理系统、卫星跟踪系统、航空电子设备等。
- 数字图像设备，如打印机、传真机、数字复印机、数字相机等。
- 网络通信中，如交换机、路由器、远程访问服务器、ATM及帧中继交换机等。
- 电信设备，如移动通信基站、蜂窝式电话等。
- 交通运输领域，如汽车发动机控制、交通信号控制、高速列车控制等。
- 工业控制领域，如机器人、测试与测量设备、过程控制系统等。
- 计算机外设，如网络计算机、RAID存储系统、I/O控制设备等。

1.6.8 μC/OS-II

μC/OS-II是著名的源代码公开的实时内核，是专为嵌入式应用而设计的。它可用于8位、16位和32位的单片机和数字信号处理器（DSP）。它在原版本μC/OS的基础上作了重大的改进和升级，并已经具备了10余年的应用实践，因此在多个领域具备了许多成功应用案例。

μC/OS-II主要有以下一些特点：

1）公开源代码，可以方便程序设计人员把该操作系统移植到不同的硬件平台上。

2）具备良好的可移植性。该操作系统绝大部分元代码是采用C语言进行编写，便于程序设计人员把它移植到其他微处理器上。

3）可以固化。

4）具备良好的可裁减性。可以很方便地选择需要使用的系统服务，这样就有效地减少了所需的存储空间。

5）完全是抢占式的实时内核，也就是说，它总是在运行条件下优先级最高的任务。

6）是个多任务系统，最多可以管理64个任务，但是这些任务的优先级必须是不同的，同时它不支持时间片轮转调度法。

7）具备可确定性，即函数的调用与服务的执行时间具有可确定性，而不依赖于任务的多少。

8）具备良好的实用性和可操作性。

第2章　嵌入式系统的软硬件协同设计

2.1　传统嵌入式系统的设计方式

本书主要介绍的是Windows Embedded CE 6.0程序设计的相关内容，但是作为一种非常重要的嵌入式操作系统，我们在开始学习Windows Embedded CE 6.0程序设计的相关内容之前，首先来看一下嵌入式系统相关的软硬件设计方式。

同设计任何软件或硬件系统一样，我们在了解清楚所设计嵌入式系统的具体需求以后，接下来就要着手进行相关软硬件的设计工作。同单纯的软件、硬件设计不同，嵌入式系统的设计是需要考虑软硬件之间的相关影响、相互制约以及如何相互补充的。嵌入式系统的软硬件设计通常包括需求分析、软硬件选择、方案设计等。而其中软硬件的选择通常包括下面几个部分：

- 处理器的选择。
- 硬件部件的选择。
- 运行操作系统的选择。
- 采用编程语言的选择。
- 软件开发工具的选择。
- 硬件调试工具的选择。
- 软件组件的选择。

同时，嵌入式系统的设计必须要考虑系统的实际应用环境，如温度、湿度、压力、灰尘等对其的影响，还要重点考虑下面几个问题：

- 系统的功耗。
- 系统的成本。
- 系统的体积。
- 系统的可靠性。
- 系统采用处理器性能。

随着嵌入式技术的不断发展，以及嵌入式硬件集成程度、可靠性、标准化等方面的不断进步，嵌入式系统的设计也越来越向系统软件的开发和应用软件的开发倾斜。如本书介绍的Windows Embedded CE 6.0就是微软公司推出的一种非常重要的嵌入式操作系统，而运行该操作系统之上的各种应用软件的开发则成为了近期Windows开发的一个热点。

传统的嵌入式系统设计方式通常采用软硬件分开设计的方法。其设计思想就是根据需求分析得出的内容，同时结合程序设计人员的设计经验，由系统分析人员粗略地把功能分配到软件和硬件中，然后再对软件和硬件分别进行设计。在软件和硬件分别实现后，再对软硬件进行集成测试，如果集成测试出现问题，再对先期的设计进行修改和完善。

图2-1展示了传统软硬件设计方式。

2.1.1 先软件后硬件的设计方式

先软件后硬件的设计方式是指先设计那些与硬件无关的软件部分，比如一些数据处算法，而那些与硬件密切相关的软件则需要等硬件确定了以后再进行设计。但是由于这种设计方式缺少了先期硬件的支撑，相当于“无源之水，无本之木”，因此缺点也是显而易见的。如后期硬件设计人员不到位，迟迟无法确定硬件所采用的处理器，接口电路选型没有确定等，这些都会极大地影响硬件的设计工作。加之这种设计方式主要被应用在算法非常复杂的场合，因此在实际的嵌入式系统开发中极少被采用，在这里也就不再加以详细地介绍了。

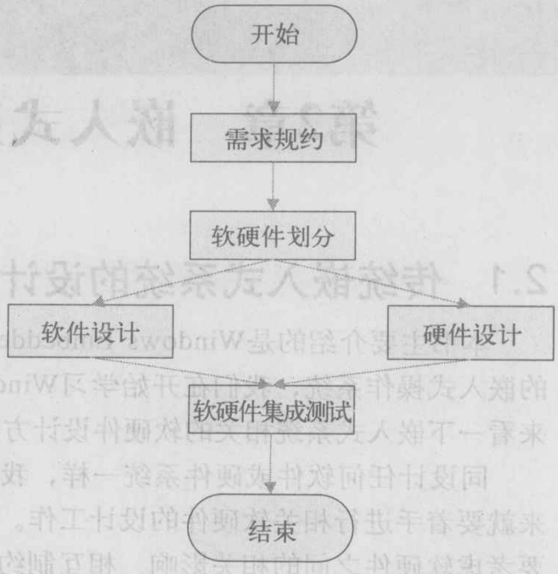


图2-1 传统软硬件设计方式

2.1.2 先硬件后软件的设计方式

在嵌入式系统的实际设计中，一般采用先设计硬件后设计软件的设计方式。即在粗略估计任务需求的情况下，首先进行硬件设计与实现；然后在此平台上，再进行软件的设计与实现。对于大多数项目来说，先做硬件设计，再做软件设计是一个非常实用的方式。实际上，这种设计思想主要是把软件作为硬件的有益补充和完善，也沿袭了早期嵌入式系统设计以硬件为主的设计方法。

先设计硬件后设计软件的设计过程一般分为以下几个阶段：

- 设计硬件子系统。
- 定义硬件接口。
- 设计软件子系统。
- 定义软件接口。
- 软硬件集成测试。

经过上面一系列步骤以后，如果达到了设计要求，则通过；否则，则返回硬件设计阶段，重新进行软硬件的系统划分。

图2-2展示了先硬件后软件的设计过程。

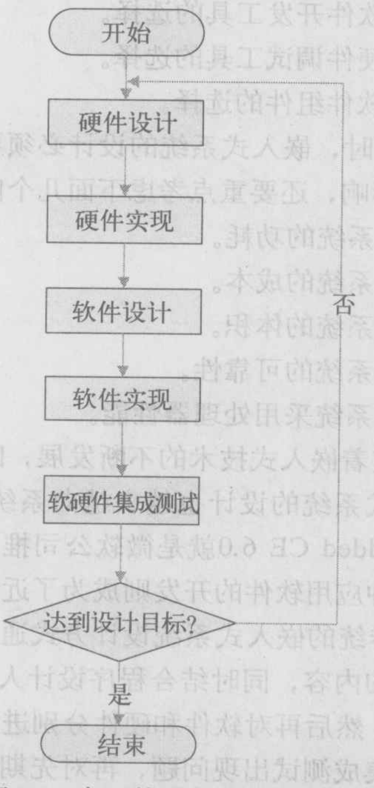


图2-2 先硬件后软件的设计方式

在先硬件后软件的设计方式中，硬件子系统的设计和软件子系统的设计是尤为重要的两个步骤。首先来看一下硬件子系统的设计。

嵌入式系统的硬件一般有以下3个来源：

- 直接选用现有的、已经成型的硬件组件。
- 对现有的硬件组件进行二次开发。

• 自己设计相应的硬件。

在这里重点介绍一下自主设计硬件的一般过程。

硬件子系统的设计一般采取“自顶向下，逐步求精”的设计方式。首先需要将硬件分成若干个部分或者模块，设计出一张或者多张硬件部件的框图，用一个框图表示一个单独的电路板或者电路板的一部分，如处理子系统、存储子系统、外设等。然后再逐步对其中各个部件的内容进行细化。

硬件功能的实现主要有下面5个关键步骤。

1）功能定义：根据系统的需求确定产品功能。

2）原理图设计：硬件设计的核心环节。在进行系统设计之前，首先就要熟悉本系统的特点，如系统的硬件指标、接口类型等。在这一步骤中设计的原理图要做到标准化、通用化、模块化、具备良好的可扩展性等。原理图设计一般会包括电源设计、复位电路设计、时钟电路设计、接口电路设计（如Flash接口、JTAG接口、USB接口、串口、并口、LCD接口）等。

表2-1展示了我们自主研发的“N_J_L_3型”（名称暂定）嵌入式开发板的硬件接口说明；表2-2展示了“N_J_L_3型”嵌入式平台的预留接口。

表2-1　“N_J_L_3型”（名称暂定）嵌入式开发板

名　称	参　数
高亮彩色TFT_LCD	800×600或320×240
单色点阵液晶（带背光）或字符液晶，高亮LED模块	可以外接黑白LED，点阵式的液晶屏
2个RS-232通信口	1个为调试/通信接口，1个为通信接口
2个USB接口	1个为USB Driver，可以实现通过USB为主板供电，并实现通信功能；一个为USB Host，可以实现USB通信功能
自适应网卡接口	10～100Mbit/s自适应网卡
键盘接口	可以外接USB或PS/2键盘
鼠标接口	可以外接USB或PS/2鼠标
打印机接口	可以接微型打印机、普通打印机
PWM输出接口	驱动直流伺服电机和混合式步进电机
触摸屏	国标4线，16bit高精度触摸屏

表2-2　“N_J_L_3型”（名称暂定）嵌入式平台的预留接口

名　称	参　数
AUDIO 双声道环绕立体声	可以根据需要添加杜比解码
工业RS-485通信口	
SD卡接口	支持1GB容量的SD卡
视屏采集卡接口	
摄像头接口	
云台数据采集	
工业CAN通信接口	博世公司专用于汽车通信的标准
智能IC卡接口	可外接相应的读卡器
标准红外接口	实现标准的红外通信功能

（续）

名　　称	参　　数
硬盘接口	国际标准的IDE硬盘接口
多通道高速工业A/D采集接口	每通道高达100MHz的采样速率
PS/2主机端控制接口	可以外接特殊功能的键盘
词条扫描	实现条形码扫描功能
语音播报模块	可以实现输入汉字，直接语音播放（外购模块）

3）PCB设计：目前的PCB板设计技术已经朝着高密度、高精度、细孔径、细导线、细间距、高可靠、多层次、高速传输、轻量、薄型等方向发展。在这里需要注意的是，在设计PCB板的时候，要注意PCB板布局的合理、系统电源入口作高低频滤波处理、时钟电路尽量靠近芯片、高速总线的走线尽量等长、差分信号线要平行等长、走线要规范等事项。

4）制版组装：在嵌入式系统的设计中，制版工作一般都是交给专业公司来完成的，但是元器件的焊接工作则大多由开发人员自己完成。确切地说，在研发阶段是由硬件设计人员自己焊接组装的，而到了生产阶段则是由专门的人员进行焊接的。

5）硬件调试：硬件调试一般包括电源调试、单独模块调试、对比调试、时钟调试、复位调试等。在这一阶段，需要我们具备严谨的工作态度和业界工作经验。

在了解了硬件子系统的设计之后，再来看看软件子系统的设计。

软件子系统的设计主要是依据硬件系统提供的环境进行。在嵌入式系统发展的早期，由于处理器速度和处理能力上的限制，导致了嵌入式软件系统不能进行复杂的、大量的数据处理。因此，在早些时候嵌入式系统的软件一直作为硬件系统完善和有益的补充。但是随着技术的不断进步，以及嵌入式技术应用的日益广泛，嵌入式系统中的软件已经由原来处于硬件的附属地位上升到与硬件设计同等重要的位置。

嵌入式系统的软件子系统通常包含下面几部分内容：

- 定义软件接口。
- 规定系统的启动和关闭过程。
- 定义异常处理方式。
- 监视计时器。

同任何事物都存在优点和缺点一样，先硬件后软件的设计方式尽管在实际设计中被大量采用，但是仍旧存在下面一些不足：

- 过分依赖系统分析员的开发经验。
- 当后期出现问题以后，需要进行硬件部分的重新设计。
- 通常情况下被应用在系统规模比较小的嵌入式系统开发中。

2.2 软硬件协同设计方式

从前面的介绍中可以看出，在传统的嵌入式系统设计中，是将软件和硬件作为两个独立的部分来进行设计的，即所谓的“先硬件，后软件”原则。由于采用了这种设计方式，导致了在硬件设计过程中缺乏对软件架构和实现机制的清晰而有效的了解，因此硬件的设计工作也带有一定的盲目性。

同时，早期的嵌入式系统设计采用的先硬件、后软件的设计模式需要反复修改、反复试验，整个设计过程在很大程度上依赖设计者的经验，因此开发成本较高，设计周期较长，经过多个过程的迭代修改以后往往会导致在某些方面出现背离原始设计要求的现象。

而软硬件协同设计正是为解决上述问题而提出的一种全新的系统设计思想。它与传统嵌入式系统设计思想的最大不同体现在“协同”这个概念上。即在系统设计的每一个阶段都是把硬件和软件放在一起进行考虑。它根据系统目标的要求，通过综合分析系统软件、硬件功能以及现有资源，最大限度地发掘出系统硬件和软件之间的并发性，协同设计系统的软件和硬件体系结构，使系统可以工作在最佳状态下。这种设计方式充分利用了现有的软硬件资源，缩短了系统研发的周期，有效地降低了开发成本，提高了系统的系统性，避免由于独立设计软硬件体系结构而带来的弊端。

下面就来看一下什么是软硬件协同设计。

2.2.1 什么是软硬件协同设计方式

所谓的软硬件协同设计就是指当我们协同设计一个时间要求很高的复杂系统时，软件和硬件两个部分是可以同时进行的。在进行软硬件系统设计时，软件设计人员必须了解硬件设计的种类和硬件设计时可能会遇到的问题，还必须了解硬件设计的可行性及其性能；同样，硬件设计人员也应该了解软件设计及实现的进度，软件对硬件的要求等。同时，软硬件协同设计方式要求在需求分析阶段，应该由软件和硬件设计人员共同参与，共同确定方案。

软硬件协同设计经过评估后的系统需求和系统设计方案需要进行软硬件划分，产生硬件描述、软件描述和软硬件接口三个部分。之后对各个部分进行详细设计、具体实现并进行优化。最后进行软件、硬件的整合、系统仿真与测试。

软硬件协同设计一个最主要的优点是在设计过程中，硬件和软件的设计是相互作用的。这种相互作用发生在设计过程的各个阶段和各个层次。设计过程充分体现了软硬件的协同性。同时，这种设计方式在软硬件功能划分的时候就考虑到了现有的软硬件资源，在软硬件功能的设计和仿真评价过程中，软件和硬件是相互支持的，这就使得软硬件功能模块能够在设计开发的早期互相结合，从而及早发现问题并尽快加以解决，避免了在设计开发的后期反复修改系统以及由此带来的一系列问题。有利于挖掘系统潜能，缩小产品的体积，降低系统成本，提高系统整体的性能。

软硬件协同设计是一种新的嵌入式系统开发思路，在其研究和生产实践过程中，提出了许多的设计模型。这些模型都是把系统功能转换成组织结构，将抽象的功能描述模型转换成组织结构模型。由于针对一个系统可以建立多个模型，因此应该根据系统的仿真和以前经验来选择模型。

软硬件协同设计流程是从目标系统构思开始的。对于一个给定的目标系统需要经过构思，完成该系统的规范描述，然后逐步进行下面几个阶段：

- 模块的行为描述。
- 模块的有效性检查。
- 软硬件划分。
- 性能评估。
- 硬件综合。
- 软件编译。

- 软硬件集成。
- 软硬件协同仿真与验证。

其中软硬件划分后会产生以下三个部分：

- 硬件部分。
- 软件部分。
- 软硬件接口。

其中硬件部分需要遵循硬件描述，硬件综合与配置，生成硬件组件和配置模块的步骤；软件部分则需要遵循软件描述，软件生成和参数化的步骤，生成软件模块；最后则需要把生成的软硬件模块和软硬件接口进行集成，并进行软硬件协同仿真，以便进行系统的评估和设计验证。

图2-3和图2-4分别展示了两种不同的软硬件协同设计流程；图2-5则展示了软硬件划分流程。

其中图2-4所示的软硬件协同设计流程对每个子过程都进行了较为详细的描述，尤其是在设计一个包含复杂算法的系统时，在确定系统说明文当之后，要先建立高级算法模型，然后再考虑软硬件的划分，这样就可以更好地分析算法和实现方法，如是用硬件实现还是用软件实现等。

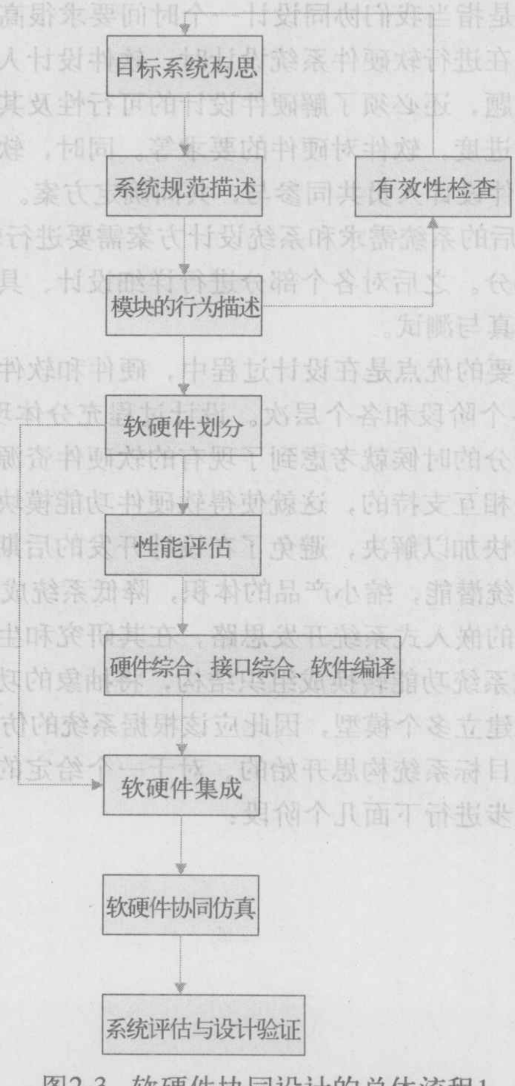


图2-3 软硬件协同设计的总体流程1

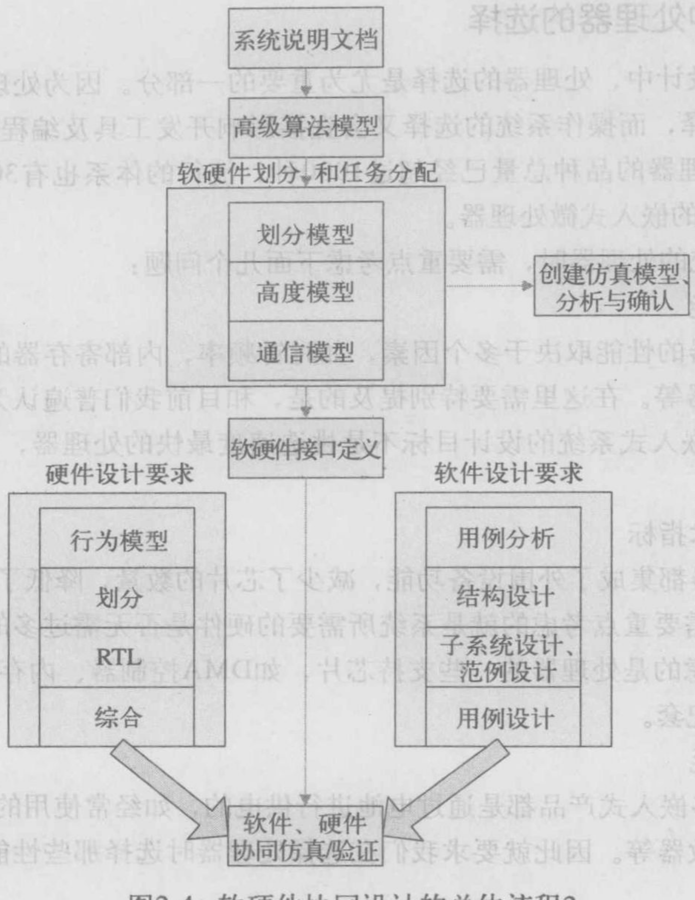


图2-4 软硬件协同设计的总体流程2

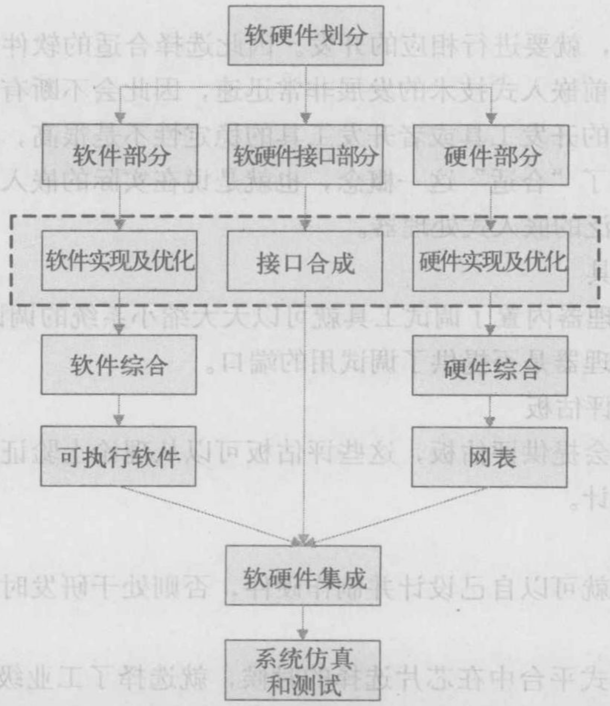


图2-5 软硬件划分流程

2.2.2 协同设计中处理器的选择

在嵌入式系统的设计中，处理器的选择是尤为重要的一部分。因为处理器的选择会直接影响和限制操作系统的选择，而操作系统的选择又会直接影响开发工具及编程语言的选择。据统计，目前在嵌入式领域处理器的品种总量已经超过1000种，流行的体系也有30多个系列，仅32位的CPU就有100多种以上的嵌入式微处理器。

在选择嵌入式系统的处理器时，需要重点考虑下面几个问题：

（1）处理器的性能

一个嵌入式处理器的性能取决于多个因素，如时钟频率，内部寄存器的大小，指令是否可以对等处理所有的寄存器等。在这里需要特别提及的是，和目前我们普遍认为的处理器速度越快越好的观念不同，许多嵌入式系统的设计目标不是挑选速度最快的处理器，而是挑选最为“合适”的处理器。

（2）处理器的技术指标

许多嵌入式处理器都集成了外围设备功能，减少了芯片的数量，降低了整个系统的开发费用。因此在这里开发人员需要重点考虑的就是系统所需要的硬件是否无需过多的逻辑就可以直接连接到处理器上。其次考虑的是处理器的一些支持芯片，如DMA控制器、内存管理器、中断控制器、串行设备、时钟等的配套。

（3）处理器的功耗

在实际应用的很多嵌入式产品都是通过电池进行供电的，如经常使用的手机、各种掌上电脑、GPS导航仪、手持播放器等。因此就要求我们在选择处理器时选择那些性能高而功耗又较低的处理器。

（4）开发软件的支持

在选择完处理器以后，就要进行相应的开发。因此选择合适的软件开发工具就会对软件开发起到良好的促进作用。目前嵌入式技术的发展非常迅速，因此会不断有新处理器涌现出来。但是往往这些处理器没有合适的开发工具或者开发工具的稳定性不是很高，因此后期会延误系统的开发。在这里也在此体现出了“合适”这一概念，也就是说在实际的嵌入式系统开发中，要尽量选择那些技术成熟，应用广泛的嵌入式处理器。

（5）是否内置调试工具

如果选择的嵌入式处理器内置了调试工具就可以大大缩小系统的调试周期，降低调试的难度。同时这里还是重点考察处理器是否提供了调试用的端口。

（6）供应商是否提供评估板

许多处理器供应商都会提供评估板，这些评估板可以从理论上验证设计是否正确，帮助我们的开发人员检查自己的设计。

（7）生产的规模

如果生产规模较大，就可以自己设计并制作硬件，否则处于研发时间和成本的考虑最好从第三方购买现成的板卡。

我们自主研发的嵌入式平台中在芯片选择的时候，就选择了工业级别的ARM9嵌入式处理器AT91RM9200。之所以选择它，是因为AT91RM9200是完全围绕ARM920T ARM Thumb处理器构

建的系统。它有丰富的系统与应用外设及标准的接口，从而为低功耗、低成本、高性能的计算机宽范围应用提供一个单片解决方案。

AT91RM9200包括一个高速片上SRAM工作区及一个低等待时间的外部总线接口（EBI），以完成应用所要求的片外存储器和内部存储器映射外设配置的无缝连接。外部总线接口EBI 有同步DRAM（SDRAM）、Burst Flash 及静态存储器的控制器，并设计了专用电路以方便与SmartMedia、CompactFlash 及NAND Flash 连接；高级中断控制器（AIC）通过多向量，中断源优先级划分及缩短中断处理传输时间来提高ARM920T处理器的中断处理性能；外设数据控制器（PDC）向所有的串行外设提供DMA 通道，使其与片内或片外存储器传输数据时不用经过处理器，这就减少了传输连续数据流时处理器的开销；包含双指针的PDC控制器极大地简化了AT91RM9200的缓冲器连接；并行I/O（PIO）控制器与作为通用数据的I/O 复用外设输入/ 输出口线，以最大程度上适应器件的配置。每条口线上包含有一个输入变化中断、开漏能力及可编程上拉电阻；电源管理控制器（PMC）通过软件控制有选择的使能/ 禁用处理器及各种外设来使系统的功耗保持最低。它用一个增强的时钟产生器来提供包括慢时钟（32 kHz）在内的选定时钟信号，以随时优化功耗与性能；AT91RM9200集成了许多标准接口，包括USB 2.0 全速主机和设备端口及与多数外设和在网络层广泛使用的10/100 Base-T 以太网媒体访问控制器（MAC）。此外，它还提供一系列符合工业标准的外设，可在音频、电信、Flash 卡红外线及智能卡中使用。

为完善性能，AT91RM9200集成了包括 JTAG-ICE、专门UART调试通道（DBGU）及嵌入式的实时追踪的一系列的调试功能。这些功能使得开发、调试所有的应用特别是受实时性限制的应用成为可能。

下面列举了这款芯片的一些特点：

- 工作在180MHz时性能可以达到100MIPS（million instruction per second，每秒处理的百万级的机器语言指令数）。
- 16KB的数据缓存，16KB的指令缓存。
- 含有调试信道的内部仿真器。
- 中等规模的嵌入式宏单元结构。
- 低功耗，VDDCORE（核电源引脚）电流为30.4mA，待机模式电流为3.1mA。
- 附加的嵌入式存储器SRAM为16KB，ROM为128KB。
- 外部总线接口支持SDRAM、静态存储器、Burst Flash，无缝连接的CompactFlash、SmartMedia及NAND Flash。
- 具备增强的时钟发生器与电源管理控制器，低速的时钟操作模式与软件功耗优化能力，四个可编程的外部时钟信号，有报警中断的实时时钟，有8个优先级高级中断控制器，独立的可屏蔽中断源。
- 具备122个可编程的I/O口。
- 10/100Mbit/s BASE-T型以太网接口。
- USB2.0全速主机双端口。
- USB2.0全速器件端口。
- 多媒体卡接口，支持SD存储器。

- 3个同步串行控制器。
- 4个通用同步/异步接收/发送器。
- 主机/从机串行外设接口。

2.2.3 协同设计中软件平台的选择

在嵌入式系统的硬件设计方案确定以后，软件平台的选择就变得相对轻松了。嵌入式系统软件的开发流程，主要涉及下面几个步骤：

1）软件编程。

2）交叉编译。

3）交叉连接。

4）下载到目标板。

5）调试。

因此，软件平台的选择主要涉及操作系统、编程语言和集成开发环境三个方面。操作系统的选择主要从下面几个方面进行考虑：

1）操作系统提供的开发工具。有些嵌入式操作系统只支持该系统供应商提供的开发工具，因此当选择这类处理器的时候，还必须选择该供应商提供的编译器、调试器等工具，导致了后期开发成本的增大和开发灵活性的降低；而有些嵌入式操作系统则可以很好地支持第三方工具，因此在开发的时候选择余地和开发灵活性就较大。

2）操作系统向硬件接口移植的难度。操作系统向硬件系统的移植是一个非常关键的问题，因此要选择那些可移植性高的嵌入式处理器。

3）操作系统的内存要求。

4）开发人员是否熟悉此操作系统及其提供的API函数。

5）操作系统是否提供硬件的驱动程序，如SD卡、LCD平面驱动等。

6）操作系统的可裁减性。

7）操作系统的实时性。

目前嵌入式系统开发中使用的编程语言种类较多，应用广泛的有Ada、C、C++、J2ME等。而在本书中则主要介绍采用C++、C+、Visual Basic程序设计语言，在Embedded Visual C++以及Visual Studio 2005集成开发环境下进行Windows Embedded CE 6.0程序设计的过程。

在选择编程语言时需要考虑下面几个因素：

1）通用性。

2）可移植程度。

3）执行效率。

4）可维护性。

集成开发环境（Integrated Development Environment，IDE）是进行开发时的重要平台，选择时需要考虑下面几个问题：

1）系统调试器的功能，包括远程调试环境等。

2）所支持的库函数。

3）编译器开发商是否能够持续升级编译器。

4）连接程序是否支持所有的文件格式和符号格式。

2.3 嵌入式系统设计时需要注意的问题

嵌入式系统设计时，需要注意以下几个问题：

1）在需求和规范分析中存在的问题。出现实时事件时，程序流程控制规范是什么？每个软件功能可以接受的最大响应时间是什么？什么是事件的最大延时？什么是中断服务例程可接受的最大延迟规范？什么是软件组件和中断优先级的规范？如何向那些具有更短时限的中断分配优先级？可靠性与可接受的容错性……

2）设计嵌入式系统硬件不只是挑选处理器。关键是如何将系统划分为硬件系统与软件系统。划分决策既不能只根据一些硬性的规定，也不能只是想当然地作出一个判断信息。而是要求我们的设计人员必须决定问题的哪一个部分应该在硬件中解决，哪一个部分应该在软件中解决。

3）设计和实现中的问题。处理器、存储器及其大小和硬件的选择。单处理器和多处理器的选择。能够使系统中存储器和功耗得到优化的软件设计，决定是否使用高速缓存以及如何。是让软件开发小组设计嵌入式操作系统还是使用已经测试和调试好的嵌入式操作系统。测试方法，调试端点以及宏的选择。

4）任何功能都可以通过硬件来实现；有些功能可以通过硬件和软件的配合实现；而有些功能则可以纯粹的软件来实现。

5）设计过程中，硬件与软件的划分与实际的实现之间仍旧存在某些模糊区域，涉及依然变化无常，还是留有大量余地来移动划分之间的界线。在这个阶段中，随着更多的设计约束被理解和建模，将进一步明确硬件与软件的划分界限。所以，软硬件的任务划分必须提供多个方案，不能只提供一个方案，对多个方案进行分析和评估，从中选择一个最能满足系统的开销、性能等方面限制的方案，即选择最优的方案。

6）如果系统规模较大或者硬件较为复杂，很可能要软硬件设计同时进行，以便加快整个项目的设计进程。软件设计者和硬件设计者需要很好的沟通，并定期地交流先前的设计决策，以确保软硬件设计保持一致。

7）系统集成中的问题。使用适当接口的统一规范的集成和开发，先对接口进行验证，然后对接口系统进行验证。

8）测试中的问题。对于实时系统，需要进行白盒测试、黑盒测试以及冒烟测试等。需要处理的问题有测试策略的选择，硬件和软件集成的系统测试，测试适当的中断优先级分配，对任务和ISR的吞吐量的测试，IPC的测试以及软件在硬件中的可移植性测试等。

9）选择合适的平台。平台是由许多单元组成的，如处理器、ASSP、片上系统、存储器、总线、嵌入式操作系统、软件编程语言、代码生成工具和系统的其他硬件单元。设计者需要选择上述单元以最终得到合适的平台和开发工具。

10）处理器的选择。使用带有微处理器、微控制器或者DSP的系统。另外，PLC操作速度很慢，计算能力也很差，但是具有非常强大的多输入/输出接口。具备系统相关的可编程能力，有些时候可以代替处理器。

11）需要考虑必须的特性和因素。使用8位或者16位还是32位的芯片，存储器的大小，同步

通信是全双工还是半双工，输入捕捉和输出比较，功耗大小等。

12）软硬件权衡。在硬件和软件之间需要权衡。硬件中的某些子系统、实时时钟、脉宽调试器、定时器和串行通信也可以通过软件实现。硬件实现不仅可以加快操作速度，也会增加开销。软件实现有利于随着新需求的出现而迅速作出改变。

13）系统性能指标。系统性能指标可以定义为，在使用最少量的存储资源、功耗和设备，最少的设计工作量，以及每个资源优化使用的情况下，满足所需功能和规范的能力。最好的嵌入式硬件是那些可以在不同性能度量之间获得平衡的软硬件。

第3章 Windows Embedded CE 6.0简介

3.1 Windows CE的发展历史

在本书中介绍的Windows Embedded CE 6.0是微软在整个嵌入式产品布局中一个非常重要的分支。微软在整个嵌入式系统中共有下面3个分支的产品：

- .NET Micro Framework。
- Windows Embedded CE 6.0。
- Windows XP Embedded。

图3-1展示了微软在嵌入式领域各个分支的主打产品。

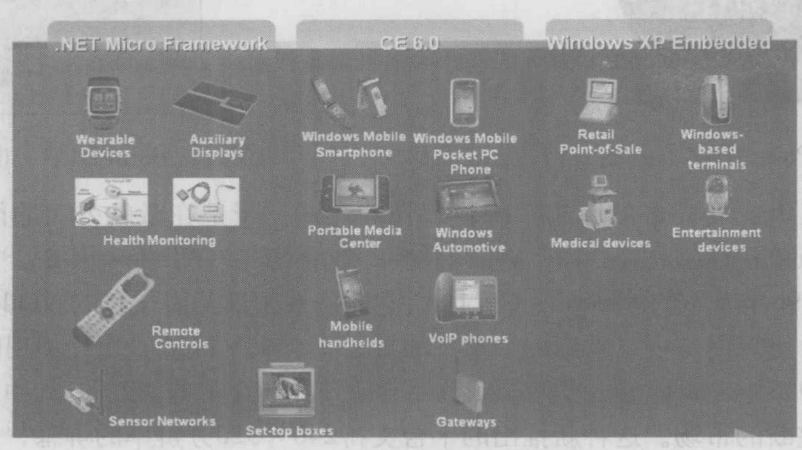


图3-1 微软在嵌入式领域的主打产品

Windows CE是微软公司在嵌入式领域一款非常重要的产品，于1996年首次发布。起初Windows CE并不是非常的成功，直到推出Windows CE 3.0，它才真正地被世人所接受。与Windows 98/2000/XP的API函数相比，Windows CE中的应用程序接口（API）是一个精简的WIN32 API，也就是说它是桌面Windows系统API的一个子集。这就使得很多原来基于桌面Windows开发的应用程序在经过很少的改动以后，就可以被用于Windows CE中。但是在某些情况下，由于Windows CE中的API与标准桌面Windows API之间存在一定的差异，这就导致开发人员需要采取别的办法来实现这些功能。同时，由于Windows CE支持的系统内存容量和显示屏幕面积非常有限，也使得开发人员不得不仔细考虑与硬件相关的因素。为了更好地了解Windows Embedded CE 6.0，首先来看它是如何产生的。

在传统PC时代，人们就一直梦想拥有一种可以随身携带的计算设备，这种设备体积小、方便携带、具备一定的计算能力、可以满足一般办公需要等功能，正是这种需求促使了掌上电脑的产生。掌上电脑的出现极大地改变了人们的生活，包括学习方式和工作习惯等。很多人因此抛弃了传统的记事本、计算器、电话本等，掌上电脑的使用率也在逐渐地提高，甚至成为了人们的生活必需品。

Apple公司于1993年推出的Newton可以算是掌上电脑的开始，因为它已经具备了现代掌上电脑的雏形。Newton功能强大，可上网，可与PC和Mac机同步，支持手写识别。但是由于速度慢、体积大、营销不力等原因，Newton并没有在市场上取得成功，最终不得不于1998年停产。

当Newton还在艰难挣扎的时候，California一个叫做Palm Computing的公司却是一片繁荣。它生产的设备价格比Newton便宜，其产品PalmPilot一跃成为历史上销售最快的电子设备。Palm和它的PalmPilot很快成为技术工业的宠儿。1998年，Palm发布了第三代掌上电脑Palm III。Palm III使用的是Palm 3.0操作系统，拥有2MB容量的内存和2MB的Flash ROM，并且支持红外传输。

图3-2和图3-3所示的是当时非常流行的Palm III掌上电脑。

图3-2　Palm III掌上电脑（1）

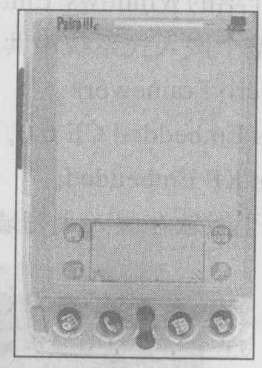


图3-3　Palm III掌上电脑（2）

掌上电脑的巨大成功，自然引起了微软公司的注意，于是在微软公司的内部就开始了名为portable PC solution的项目，该项目的名称有很多，最后更名为Windows CE。

微软公司于1996年发布的Windows CE 1.0，它的外观很像当时流行的Windows 95的界面。这样做的好处就在于使得用户可以看到一个熟悉的界面，大大地降低了用户使用时的难度，使其更容易被接受。在1998年，微软推出了两个新的平台：掌上电脑和车载PC。其中掌上电脑直接瞄准了被Palm公司所垄断的市场。这种新推出的平台支持240 × 320分辨率的屏幕，同时使用手写笔进行屏幕的操作。但是非常不幸，大家对于这种非常原始的掌上电脑缺乏热情，因此导致了市场上的失败。在2000年4月，微软公司推出了基于Windows CE 3.0核心的新型的掌上电脑。这种设备凭借低廉的价格、大屏幕和持久的电池使用时间在工业控制领域赢得了市场。2001年年底，微软公司又推出了Pocket PC2002，这次采用的是Windows CE 3.0的最终发布版本，在用户接口方面也进行了一些必要的改进，在此基础上还添加了电话模块，使得这些设备不但具备掌上电脑的特性，同时也具有了电话功能。2003年春季，Pocket PC团队发布了升级版的Pocket PC——Pocket PC2003。这个系统在用户界面方面没有多大的变化，同时由于它是基于Windows CE.Net 4.2的，因此在系统的稳定性和产品性能方面都取得了巨大的进步。

在Windows CE.Net 4.2取得成功的基础上，微软公司在2005年推出了Windows CE 5.0。在这个版本的Windows CE中，系统首次支持了Direct3D，这就意味着基于Windows Mobile的手持设备在游戏和多媒体上有了更大的发展空间。可以说，Windows CE 5.0的推出，使得微软在嵌入式领域的地位又向前推进了一步。因为就商业经营来说，Windows CE 5.0是一款非常成功的产品，虽然偏弱的性能表现广为使用者所诟病，但是其广泛的应用程序支持及多媒体表现却又让它成为消费者的最爱。Windows CE 5.0与前一版操作系统的差别主要体现在储存能力以及文件系统方面

的改进。对于OEM制造厂商来说，Windows CE 5.0最占优势的一项改进，就是QFE快速修复功能的增强，透过QFE机制，当嵌入式设备需要进行功能更新或者除错时，生产厂商无须重新发布完整ROM内容，只需让用户下载特定部分的组件，然后更新即可。这样做不但提高了更新的效率，还把由于系统更新而导致的资料损失降低到最低的限度。

在针对便携式智能设备的Windows Mobile 5.0中，更是增加了几个重大的更新，其中之一就是加入了对3G网络的支持。虽然3G的推广受到应用不足以及价格高昂的阻碍，距离大规模普及还有一定的距离，但是其可进行高速网络传输的能力，对特定商务或消费人群仍有相当大的帮助，因此未来潜力仍不可小视。Windows Mobile 5.0另外一个大的变化体现在文件保存方式及文件的执行方式上。不同于以往的Windows CE，Windows Mobile 5.0将原先仅设计作为储存之用的ROM也拿来给程序执行时使用，因此资料的储存就不需要用额外的电力去维持，除了省电以外，也可以确保当系统突然断电时，资料仍可以完整被保留住而不会消失。

正如任何事物都有两面性一样，上面提到的做法确保了资料的安全性，却带来了性能上的极大耗损。由于智能设备所使用的ROM是属于读取速度快但写入速度慢的存储器，因此在程序执行阶段其性能要比老版本的Windows CE差。尽管相差的幅度还不是很大，但是当Windows Mobile 5.0要进行分页动作或者要将资料写回ROM中时，设备就会表现出明显的性能滞后。有时系统不仅无法回应使用者的操作，程序的执行有时也会被迫中断。

在传统计算机领域，源代码开放吸引了越来越多的年轻人，也撼动了微软产品一统天下的局面。而在嵌入式领域，微软同样遇到了以嵌入式Linux为代表的开源软件的大力追杀。因此微软在推广这一版操作系统时，开放了部分源代码。不过Windows CE 5.0只开放了约56%的源代码，这也使得Windows CE 5.0刚推出时造成了不小的争议。

Windows CE 5.0另外一个不得不提到的变化就是增加了对精简版DirectX的支持。Windows CE 5.0平台可以提供相应的DirectX8和DirectX3D硬件加速能力，这给导航设备画面显示、应用程序的图形界面，或者游戏开发等都带来了崭新的视觉体验，极大地提高了用户的使用感受。

伴随着移动计算相关技术的不断进步，以及移动设备应用的日趋广泛，微软公司又在2006年推出了Windows Embedded CE 6.0，这也是目前最新版本的Windows CE。

在旧版的Windows CE中，同时间只能有32个程序执行于各自分配的32MB虚拟存储空间中，而Windows Embedded CE 6.0则大幅放宽了限制，最大可同时执行32 000个程序，而且每个程序可拥有独立分配的2GB虚拟存储空间。在此同时，核心服务、硬件的驱动程序、窗口绘图以及事件子系统、文件系统等服务都被转移到系统核心空间中。同时，Windows Embedded CE 6.0依旧把重点放在ARM架构中，新的BSP与编译器也都加入了对ARM最新体系的支持。

在Windows Embedded CE 6.0中，微软首次提出了ExFAT文件系统这一概念。ExFAT在Windows Embedded CE 6.0中，担当了管理所有外接储存的中界层的角色。它一改传统FAT文件系统32GB单一容量的限制，同样也改变了单一文件只能在2GB以下的限制，这对于硬件厂商以及Windows CE高性能应用都具有相当大的帮助。

VoIP也是Windows Embedded CE 6.0另一个持续加强的重点，在Windows Embedded CE 6.0中，系统直接支持了802.11i、WAP2、802.11e（无线QoS）、蓝牙A2DP/AVRCP的AES加密等，为无线通信建立了一个稳定、安全以及可靠的应用环境。

而从用户角度来看，Windows Embedded CE 6.0还更新了多媒体方面的功能。诸如加入了对

Windows Media 10/11的支持，在Platform Builder for CE 6.0开发工具中，也加入了对便携式播放器的支持，并且借助其强大的多媒体文件处理性能，系统可以很好地与其他微软操作系统或硬件装置进行无缝的连接。这些功能主要包含以下几项内容：

- 对TIFF编解码器的支持。
- 对HD-DVD解码器的支持。
- 对MPEG-2解码器的支持。
- 对更多的影音编码与格式支持。
- 对UDF 2.5格式的支持。
- 对虚拟环绕声道的支持。
- 对多轨音效的支持。
- 强化了DirectDraw，可支持电视使用的交错显示模式。
- USB OTG功能加入，可作为USB的控制端。

虽然在核心部分做出这么大的更新，但是Windows Embedded CE 6.0在系统“体积”上并没有如微软其他操作系统般的飞涨（Vista甚至需要超过10GB的初始储存安装空间）。相比起5.0版、6.0版本在体积上也不过增加了5%左右。

正是由于微软公司对Windows Embedded CE 6.0做了如此重大的升级，因此在不改变原有硬件架构的基础上，导入Windows Embedded CE 6.0可以大幅改善原有程序的执行效率，并且也容许同时间有更多程序同步执行。同时，由于每个程序都具备有独立的执行空间，当某一程序出现特定错误的时候，也不会影响到其他应用程序或系统的运行，因此Windows Embedded CE 6.0具备了更高的稳定性。正如任何新的技术最终都需要经过市场的检验一样，目前Windows Embedded CE 6.0在物流、仓储管理、公共服务、RFID运用、GPS导航仪等方面都有很好的应用。

至于Windows Embedded CE 6.0中CE名字的由来，已经没有确切的资料可以查询。但是根据各种资料上的说法，大家都认为C代表了消费（Consumer）、袖珍（Compact）、通信能力（Connectivity）和伴侣（Companion）；E则代表了电子产品（Electronics）。这都说明了Windows Embedded CE 6.0是微软公司设计的一款全新操作系统。随着计算机技术的不断发展和消费类电子产品的迅速普及，Windows Embedded CE 6.0必将迎来一个崭新的、高速的发展空间。

图3-4展示了Windows CE的发展历程。

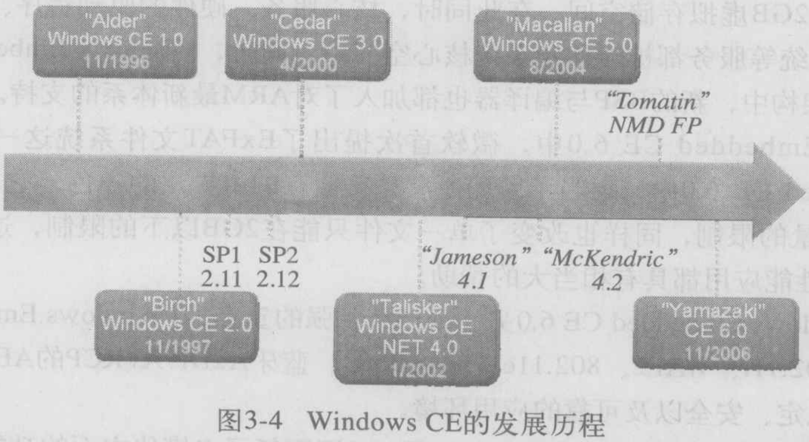


图3-4 Windows CE的发展历程

3.2 Windows CE 功能简介

Windows CE是一种具有抢先式多任务功能与强大通信能力的嵌入式操作系统，也是微软公司专门为移动设备和消费类电子产品、嵌入式应用等非PC领域而设计的全新操作系统产品。因此根据其应用环境的特点，Windows CE被设计成具有高度模块化、良好实时性、强大通信能力、支持多种CPU的嵌入式操作系统。下面就一起看一下Windows CE具有哪些基本的功能。

1. Windows CE的电源管理功能

由于嵌入式系统很大一部分需要依靠电池进行供电（如掌上电脑、智能手机等），因此Windows CE具有灵活的电源管理功能，包括睡眠/唤醒模式。在Windows CE中还使用了对象存储技术，包括文件系统、注册表以及数据库。同时，作为操作系统的一种，Windows CE也具备了一些高性能、高效率操作系统的特征，如按需换页、共享存储、支持大容量堆等。

2. Windows CE的多任务功能

使用过Windows的用户一定知道，Windows是一款多任务的操作系统。可以把Windows CE看做是Windows的简化版本，自然Windows CE也支持多任务功能。可以在Windows CE中同时执行多个任务，并在这多个任务之间来回切换。可以通过熟悉的方式来操作Windows CE，如在Pocket PC 2003中也带有精简版的Pocket Word和Pocket Excel等。可以一边用其进行文字办公，一边用系统自带的Media Player听音乐，完全可以拥有在Windows上的用户体验。

3. Windows CE的图形界面

Windows之所以取得巨大的成功，其原因之一就在于采用了友好的用户界面。用户在使用的时候，完全可以避免使用晦涩难懂的命令行代码来进行操作，只需要用鼠标在相应程序的图标上点击即可，大大方便了用户。作为微软公司的主要产品之一，Windows CE自然也具有上述特点，它的用户图形界面同样十分出色，如它拥有微软的Internet Explorer浏览器和图片浏览器等。开发人员可以利用灵活丰富的控件在Windows CE环境下为嵌入式设备开发各种专门的图形用户界面。Windows CE根据嵌入式设备的特点，还支持手写识别、声音识别等特殊功能。

图3-5和图3-6展示了Windows Embedded CE 6.0的图形用户界面。

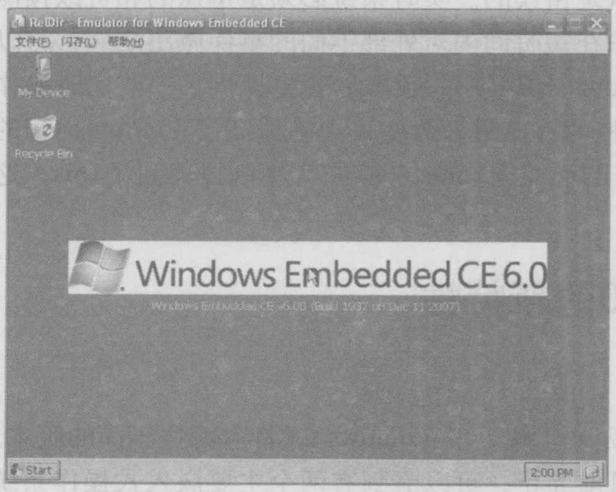


图3-5 Windows Embedded CE 6.0 图形用户界面（1）

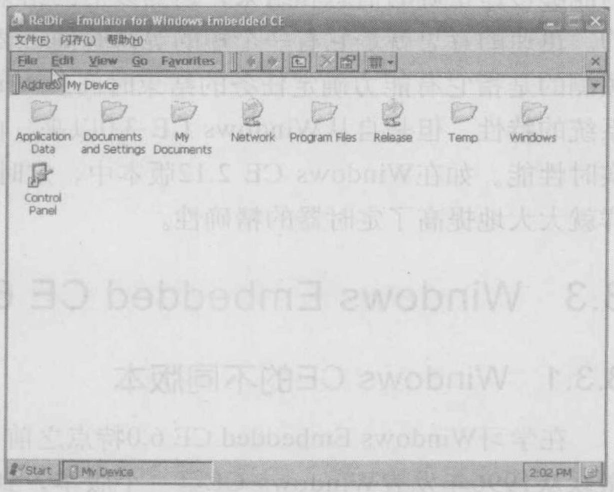


图3-6 Windows Embedded CE 6.0 图形用户界面（2）

4. Windows CE的通信

Internet的广泛应用，促进了计算机设备通信能力的提高。作为嵌入式操作系统中重要的一员，Windows Embedded CE 6.0自然也具备了非常出色的通信能力。它支持各种通信硬件、局域网连接和拨号连接，可以用于各种级别的数据传输和设备之间的连接。由于Windows Embedded CE 6.0也是微软公司的产品，因此它具有和Windows XP、Vista等桌面操作系统良好的通信能力。

5. Windows CE的多媒体和扩展功能

Windows Embedded CE 6.0在内置了Word和Excel的同时，还内置了多媒体播放器。可以通过Windows media player播放mp3音乐和.avi格式的视频。同时，Windows CE还根据其自身的特点，在机器上配置了可以扩充的MMC、SD和CF卡插槽。这样就可以通过扩展卡无限扩展Windows CE系统的功能，如添加SD卡摄像头、CF卡无线网卡和CF卡收音机等。在实现功能扩展的同时，也可以实现存储容量的扩展。目前在市场上已经可以很容易地买到容量为2GB的CF卡和SD卡。这些扩展功能在一定程度上解决了困挠嵌入式设备多年的存储容量小的问题，为嵌入式设备提供了无限广阔的市场前景。

图3-7所示为4GB容量的高速CF卡，图3-8所示为4GB容量的高速SD卡。

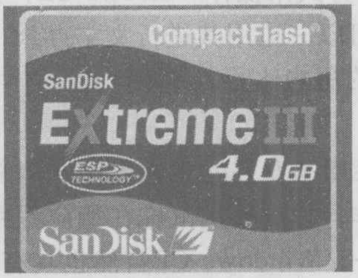


图3-7 4GB容量的高速CF卡

图3-8 4GB容量的高速SD卡

6. Windows CE的实时功能

实时操作系统（Real-Time Operation System，RTOS）中“实时”的含义就是在规定的时间内能够传递正确的信息和结果，迟到以后的结果就是错误的。实时系统必须是迅速的和可以预测的。迅速的意思就是它有一个短的等待时间，要求在短时间内对外部不同的事件做出响应；可以预测的是指它有能力确定任务的结束时间。Windows CE早期的版本尽管也提供了一些实时操作系统的特性，但是自从Windows CE 3.0以来，内核的许多重大改变极大地提高了Windows CE的实时性能。如在Windows CE 2.12版本中，定时器的性能为25ms，而在3.0中则提高到了1ms，这样就大大地提高了定时器的精确性。

3.3 Windows Embedded CE 6.0的新特性

3.3.1 Windows CE的不同版本

在学习Windows Embedded CE 6.0特点之前，先来了解一下Windows CE都有哪些不同的版本。微软从1996年发表Windows CE第一个版本，到目前为止，可以看到有两个主要嵌入式操作系统品牌。其一是针对移动终端的操作系统Windows Mobile。在Windows CE 5.0 版本之前，Windows Mobile又分为以下三种：

- 针对智能手机的版本Windows Mobile for SmartPhone。
- 针对PDA 手机的版本Windows Mobile for Pocket PC Phone。
- 针对PDA 的版本Windows Mobile for Pocket PC。

其二是微软的嵌入式操作系统平台，即Windows Embedded。这个平台主要包括以下三个部分：

- Windows CE。
- Windows Embedded XP。
- Windows Embedded for POS。

其中Windows CE可以支持各种便携设备和广泛的嵌入式应用；而嵌入式XP即Windows Embedded XP（简称XPE）则属于模块化的XP版本，它支持各种嵌入式应用，嵌入式POS系统（简称WEPOS）则是一个专门为零售终端定制的嵌入式XP版本。

如果从内核角度进行划分，微软这些嵌入式操作系统里面实际上只有两个内核，即Windows CE和Windows XP核心。

上面提到的Windows Mobile都使用的是Windows CE内核，如在目前的消费类电子产品中采用的都是基于Windows CE 5.x 版本的Windows Mobile 5.0 和6.0。而Windows Embedded XP和Windows Embedded for POS则使用了Windows XP的核心。

从微软的嵌入式操作系统分类中可以看到其在嵌入式系统上采取的策略，即Windows CE 和Windows Mobile 是具有硬实时的嵌入式操系统，目标是移动和通用的嵌入式设备，如手机、导航、PMP、机顶盒、工业控制设备和医疗仪器等；Windows Embedded XP 则是一个非实时的可以嵌入的操作系统，目标是瘦客户机、零售机器、工厂生产线控制和技术外设存储和显示设备。另外，微软最新推出的.NET Micro Framework是针对微型设备和单片机市场的一个新产品，它补偿Windows CE不能支持的更小型的嵌入式应用。

3.3.2 Windows Embedded CE 6.0的特点

同前一个版本Windows CE 5.0相比，Windows Embedded CE 6.0发生了很大的变化，这种变化不仅体现在名称上多了一个Embedded，更表现在进程数量、虚拟内存、开发工具等诸多方面。下面就来具体看一下。

1. 同时运行进程数量的增加

在Windows CE 5.0及以前的各个版本中，Windows CE系统能够同时运行的进程数量为32个。这32个进程中包含了提供系统服务的NK.exe和提供对象存储等服务的FileSys.exe两个系统进程，还包括系统常用图形界面GWES.exe进程，加载和管理设备驱动程序的Device.exe进程，提供服务管理的Service.exe进程和网络浏览器Explorer.exe进程。除去这几个进程以后，可供非系统进程使用的进程数量还剩26个。对于一个嵌入式系统来说，26个进程已经基本上可以满足实际要求，但是在目前网络环境和分布式计算广泛运用的情况下，有时26个进程就显得捉襟见肘了。因此到了Windows Embedded CE 6.0中，可以同时运行的进程数量上升到32 000个，这就解决了进程数量受到限制的问题。

2. 虚拟内存空间的增加

我们知道在Windows CE中，程序对内存的访问只能通过虚拟地址来实现，同时Windows CE还是32位的操作系统，所以它具有4GB大小的虚拟地址空间。这4GB的虚拟地址空间中，又被分

成了两个部分，其中一个为内核空间，另外一个是用户空间。在Windows CE 5.0中，用户空间被分成了64个槽（slot），每个槽32MB大小，同时一个进程只能有一个槽。而到了Windows Embedded CE 6.0中，由于其采用了新的存储机制，因此使得每个进程可以使用最大2GB的虚拟内存。

3. 去除了共享内存空间

在以前版本的Windows CE中，每一个进程都有32MB虚拟内存的限制，因此为了解决这一限制引起的问题，微软提出了共享内存空间（Shared Memory Area）的概念。所谓共享内存空间，就是指这一区域内的“空间”所有进程都可以共享，在Windows CE 5.0中，这一区域有359MB大小。而到了Windows Embedded CE 6.0中，由于每个进程都有2GB大小的虚拟内存可以使用，因此共享内存空间这一概念就完全没有必要存在了。

4. 100%公开了源代码

在Windows CE 5.0中，微软公司公开了70%的核心源代码，而到了Windows Embedded CE 6.0中，这一比例提高到了100%。

5. 提供了对VOIP的支持

在Windows CE 5.0及以前的各个版本中，系统使用TUI（Telephone User Interface）来管理与话音通信有关的服务，而到了Windows Embedded CE 6.0中，系统则使用IP Phone Suit来实现对VOIP的支持。

6. 推出了新的开发工具

在Windows CE 5.0中，使用的开发工具为Embedded Visual C++、Embedded Visual Basic和Visual Studio.net 2003，内核定制工具则使用Platform Builder。而到了Windows Embedded CE 5.0中，开发工具为Visual Studio 2005，内核定制工具Platform Builder不再作为一个独立的软件存在，变成了Visual Studio 2005的一个插件，即Platform Builder for CE 6.0。

3.3.3 Windows Embedded CE 6.0的体系结构

我们知道Windows CE 是一款典型的嵌入式操作系统，它具有层次化和模块化的体系结构。在Windows CE 5.0中，系统一般分为下面四个清晰的层次：

- 硬件；
- OEM（委托制造）；
- 操作系统；
- 应用软件。

这里提到的硬件层指的是Windows CE 可以支持的不同的微处理器和外设，如x86、ARM、XScale 等；OEM层是指引导程序（boot loader）、设备驱动等；操作系统层由内核模块、图形模块、文件和存储模块、设备管理和加载系统的服务模块组成；应用软件层则是Windows CE自身的应用软件，如MS Office、Media Player、IE和第三方应用软件。其中应用软件层和操作系统层有一个Win32 本地API和基于.Net Compact Framework的被管理代码。

而到了Windows Embedded CE 6.0中，系统则被划分成了用户模式（User Mode）和内核模式（Kernel Mode）两个“层”。被所有用户进程都加载的动态链接库CoreDLL.DLL会同时出现在两个层中。需要特别提及的是，在Windows Embedded CE 6.0中，应用程序不能直接与操作系统或硬件

打交道，如果用户程序希望访问Windows CE所提供的服务，那么只能通过CoreDLL.DLL进行。

图3-9展示了Windows Embedded CE 6.0的体系结构和包含的基本模块。

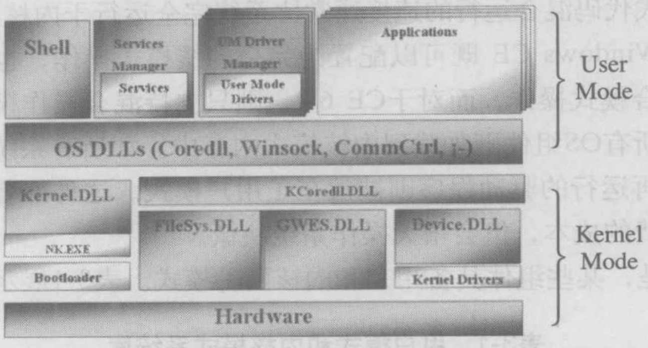


图3-9 Windows Embedded CE 6.0的体系结构和包含的基本模块

从图3-9中可以看到，原来Windows CE 5.0中的.exe可执行模块基本上都变成了.dll模块，实际上这是因为Windows Embedded CE 6.0的系统结构变化相对较大造成的。如在Windows CE 5.0中的Filesys.exe、Device.exe、GWES.exe等，都变成了Filesys.dll、Device.dll、GWES.dll，只有NK.exe还是原来的名字。之所以这样做，是因为在Windows Embedded CE 6.0中这些服务已不再是一个个单独进程，而是一个个系统调用。虽然NK.exe的名字没有变，但已不再是Windows CE 5.0中的NK.exe了，Windows CE 5.0中NK.exe提供的各种功能将由Kernel.dll来替代，NK.exe中仅仅包含一些OAL代码和确保兼容性的程序，这样做的好处是使得OEM和ISV厂商定制的代码和微软提供的Windows Embedded CE 6.0的代码进行了分离，使得内核代码的升级更加容易且更加方便。

3.3.4 Windows Embedded CE 6.0的内核结构

在本书中介绍的Windows CE也是一种微内核操作系统，这也是目前嵌入式操作系统普遍采用的较为先进的内核技术。例如，VxWorks和QNX（一种商业用途的，符合相应规范标准的类UNIX操作系统，主要被应用在嵌入式系统中）都实现了微内核技术。

所谓的微内核技术是指在内核里面只实现一些基本服务，如进程调度、进程间通信和中断处理等，其他的服务和功能都放在内核外。显然，微内核的好处体现在易于移植到不同的处理器和硬件平台，内核外的服务如设备驱动和文件管理模块是运行在不同的地址空间的，这样相较于整个系统都是平板结构的实时内核（如μC/OS-II）要更加安全和可靠。微内核的核心也非常小巧，一般几千字节到几十千字节。当然事物永远是辩证的两个方面，微内核系统因为要经常在内核态和用户态之间转换，所以系统的某些性能和实时响应能力会受到一定的影响。

同桌面Windows一样，Windows CE的每个运行程序都是一个进程。在Windows CE 5.0 版本中，系统支持32个进程，每个进程有32MB的虚拟地址空间；而最新版本的Windows CE 6.0 则可以支持3.2万个进程，且每个进程有2GB的虚拟地址空间。同时，Windows CE 还是一个基于抢占的多线程操作系统。在线程这一级，Windows CE 可以实现任务的调度、通信、同步功能。为了支持可以抢占的硬实时调度，Windows CE 还实现了优先级反转机制（priority inversion）。

Windows Embedded CE 6.0版本的WinCE内核较以前的Windows CE 5.0有了很大的改进。我们知道设备要运行 Windows CE，CPU 必须支持两个级别的权限。其中较高级别的权限称为内核模

式，较低级别的权限称为用户模式。为代码或数据分配的任何内存都将被分配给这两种模式之一。将代码置于用户模式有助于整个环境更可靠、更安全地运行。然而，这些益处的获得是有代价的，因为用户模式和内核模式代码混合运行的速度通常比系统完全运行于内核模式下时要慢一些。

在以前的版本中，Windows CE 既可以配置为完全内核模式操作，也可以配置为同时使用内核模式和用户模式的混合模式操作。而对于CE 6.0，则只支持混合操作模式，所有应用程序将加载到用户模式内存中，所有OS组件则加载到内核模式内存中。如所有系统的驱动程序被放在了内核模式，系统启动以后再运行的驱动程序则被放在了用户模式。这种部署方式有助于最大程度降低跨权限边界调用所花费的成本，但会增大操作系统映像。

在这里需要注意的是，某些组件具备用户和内核两种模式。表3-1展示了部分这样的组件。

表3-1 用户模式和内核模式系统库

组 件	用户模式	内核模式
核心系统库	coredll.dll	k.coredll.dll
设备驱动程序帮助程序	ceddk.dll	k.ceddk.dll
凭据提供程序	credsrv.dll	k.credsrv.dll
IP 网络帮助程序 API	iphlpapi.dll	k.iphlpapi.dll
多媒体计时器	mmtimer.dll	k.mmtimer.dll
进程、线程和内存帮助程序	toolhelp.dll	k.toolhelp.dll
Windows Sockets	ws2.dll	k.ws2.dll
Windows Sockets 服务提供程序接口	wspm.dll	k.wspm.dll

图3-10展示了Windows Embedded CE 6.0的内核结构。

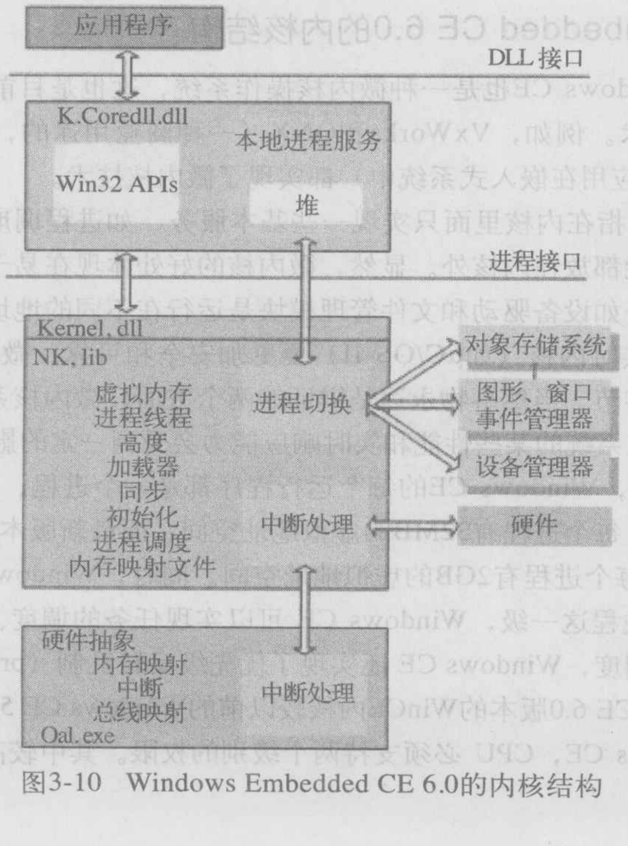


图3-10 Windows Embedded CE 6.0的内核结构

3.3.5 Windows Embedded CE 6.0的BSP

BSP是英文Board Support Package的缩写，即板级支持包。板级支持包的主要作用是对各种板子的硬件功能提供了统一的软件接口，它包括硬件初始化、中断的产生和处理、硬件时钟和计时器管理、局域和总线内存地址映射、内存分配等。实际上，嵌入式设备中的BSP和计算机中的BIOS有一些类似，两者对比学习可以加深对BSP的理解。计算机中的BIOS是硬件和软件之间的一个接口，同时也是固化到计算机存储器中的一段程序，它主要提供以下一些功能：

1）自检及初始化计算机。

负责计算机的启动，如加电自检、初始化和引导三个步骤。其目的是完成计算机硬件的检查，创建中断向量，设置中断向量表等工作。

2）程序服务处理及硬件中断处理。

主要是负责与计算机输入/输出相关的操作，如硬盘的读写、与外接设备进行数据传输等。

3）程序服务请求。

主要作用是调用中断处理来实现硬件的某些功能，并将这些功能以分组的方式配以中断号，当我们需要某种功能时，可以对其直接调用。

类似于BIOS，BSP的主要功能为：

（1）系统初始化

系统的初始化过程可以分为三个过程，分别为片级初始化、板极初始化和系统级初始化。其中片级初始化主要负责对处理器进行初始化操作。在这一过程中，需要对处理器的寄存器进行设置，同时需要调整处理器的核心工作模式和局部总线模式，其目的是把处理器从默认工作状态调整为嵌入式系统所需要的工作状态，是一个纯粹的硬件初始化过程。板级初始化是指初始化除处理器以外的所有硬件设备，同时对其他所需软件的数据结构和参数进行设置，为后面的系统初始化建立相应的软件和硬件环境。而系统初始化则是指对操作系统进行一定的初始化工作，为所运行的应用程序建立一定的运行环境，同时把系统的控制权转交给应用程序入口。

（2）提供与硬件相关的设备驱动程序

在这一过程中，BSP会把设备驱动程序与嵌入式系统中的通用驱动程序关联起来，当某个应用程序需要某个硬件设备时，通用驱动程序会调用硬件设备驱动程序，来实现操作硬件的目的。

在这里需要特别提及的是，在实际应用中，各种嵌入式操作系统是运行在不同的微处理器上的。如手机和移动设备大量使用ARM体系结构的CPU，市场上有三星ARM2410/2430、TI OMAP730/1710/2430和Marvel XScale体系的PXA270等。除此之外，数字电视、IP机顶盒等设备则使用MIPS体系结构，如东芝、博通、IDT等公司的芯片。当然，x86在各种通用嵌入式系统方面也有大量的应用。

微软的嵌入式操作系统也是使用BSP的概念来支持各种CPU和硬件平台的移植工作，在Windows CE 5.0里已经有包括三星2410、Marvel PXA270等许多流行的BSP，微软的OEM厂商如研华、飞思卡尔、NXP等也提供他们移植的OEM BSP。而最新推出的Windows Embedded CE 6.0则在Windows CE 5.0支持ARM V4的基础上支持ARM V6，同时包括三星、飞思卡尔的ARM11核在内的SoC（System-on-Chip，片上系统，即在单个芯片上集成一个完整的系统，对所有或部分必要的电子电路进行包分组的技术）都将得到良好的支持。在x86方面，微软依托在桌面系统的强势，得到了众多IPC（工业PC）和EPC（嵌入式PC）厂家的拥戴和支持。

在BSP结构方面，新的Windows Embedded CE 6.0的内核和OAL（OEM Adaptive Layer）是完全独立的两个模块，好处是在修改BSP后，内核不需要重新构建了，减少了多次构建、测试和发布内核的过程，提高了系统的可靠性。这种结构还可以让微软发布针对OEM的内核，而OEM可以因为自身知识产权的考虑，以二进制方式向最终用户发布BSP。Windows CE自身包含OEM BSP 定制和发布工具，这样OEM可以很方便地发行自己的BSP SDK包。

图3-11展示了Windows Embedded CE 6.0的内核和BSP的结构。

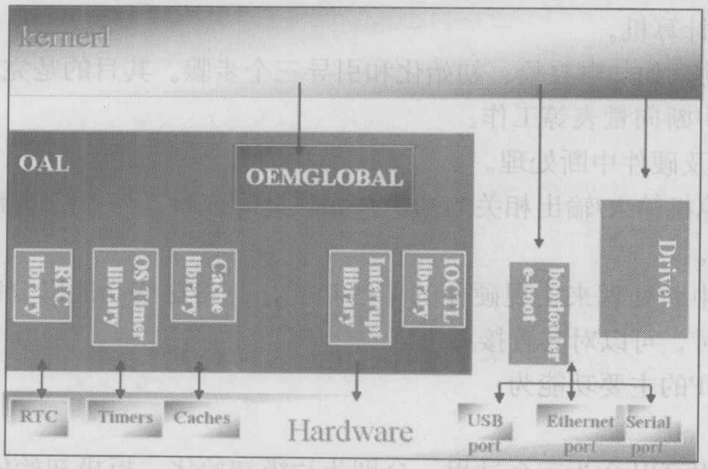


图3-11 Windows Embedded CE 6.0的内核和BSP的结构

3.3.6 Windows Embedded CE 6.0的设备驱动程序

设备驱动程序是嵌入式操作系统的一个重要部分，它是外设和应用软件的接口。在Windows CE 5.0中，所有的驱动程序全部是采用动态加载方式，即在操作系统内核启动以后对硬件外设（如USB、LCD）加载，驱动程序依然运行在内核空间。简单来讲，这种驱动程序处在用户态，它和我们熟悉的Windows 系统的DLL方式没有区别，其好处是程序员调试一个驱动的时候会很方便，和应用程序的调试没有差异，可以使用正常的函数调用，把驱动的动作和数据显示在屏幕上。同时，由于用户自己开发的驱动程序可能有不稳定和可靠的地方，因此驱动程序运行在用户态，整个系统将会更稳定些。

但是这种调用方式的问题和缺陷也是显而易见的。因为驱动程序的动态加载和内核的切换往往伴随着性能的损失，这对于桌面系统则可能没有太大的影响。但是在嵌入式系统，用户对实时性能的要求要严格得多。为了解决这个问题，在新发布的Windows Embedded CE 6.0里面，驱动程序将分为内核模式和用户模式两种方式，即把与系统直接相关的驱动程序从用户模式中解放出来，直接放在了内核模式。其中内核模式侧重效率，用户模式侧重稳健和可靠。

3.3.7 Windows Embedded CE 6.0对移动电话的支持

为了实现计算机之间的通信，Windows Embedded CE 6.0提供了连接移动电话网络所需的接口函数。在早先几个版本的Windows CE中是不支持拨打电话和发送 SMS（文本）消息的。而在Windows Embedded CE 6.0中，Microsoft提供了一组用于连接移动电话网络的精选组件。

在Windows Embedded CE 6.0 中提供了cellcore.dll，这个动态链接库进一步扩展了Win32 API

函数，其作用是支持各种移动电话服务，如启动数据连接、发送SMS消息等。

Windows Embedded CE 6.0所提供的另一个重要组件是无线接口层（RIL）驱动程序ril.dll。该组件为应用层与移动电话硬件的连接提供了低级别接口。有了这个接口，要想在Windows CE上构建移动电话程序，就不用再自己开发相应的接口层了。

Windows Embedded CE 6.0的其他支持元素还包括用于无线应用协议（WAP）的低级别组件，其中包括内核模式驱动程序（wapdrv.dll）和用户模式API（wap.dll）。在这里需要注意的是，Windows Embedded CE 6.0所提供的浏览器是一些仅支持HTML的浏览器。

在计算机实现移动之前，人们对计算机位置的确定兴致索然。一旦大型计算机进入配有空调的机房，就无须再考虑其位置。但是随着袖珍系统和车载自动移动计算机的出现，人们突然对计算机位置这一主题倍加关注。微软公司早在 Windows Mobile 5.0中就引进了对全球定位系统（GPS）API的支持。现在，在最新的Windows Embedded CE 6.0 中也提供了类似的API函数。

3.4 Windows Embedded CE 6.0的应用领域

Windows Embedded CE 6.0是微软最新发布的嵌入式操作系统。在Windows CE 5.0时代，微软公司把Windows CE的应用分为移动计算和嵌入式两大领域。作为最新版本的嵌入式操作系统，尽管在功能和系统结构上有了一定的变化，但是在这里仍旧按照以往微软对其应用领域的划分来加以介绍。

首先看一下Windows Embedded CE 6.0在移动计算领域的应用。

3.4.1 在移动计算领域的应用

移动计算领域主要包括智能手机、掌上电脑等各种移动设备。这些设备一般可以同时实现语音和数据处理两种功能。为了更好地对市场进行细分，在高端智能手机领域占有一席之地，微软公司推出了以Windows CE为核心的多种智能手机。这些手机除了可以实现基本的语言通话、短信发送功能以外，还可以实现对Word、Excel文件的浏览，可以使用Media Player播放MP3、使用Pocket Internet Explorer浏览网页等功能，一经推出就得到了市场的认可。目前，以Windows CE为核心的智能手机已经在高端智能手机领域占据了全球第2（第1为Nokia）的市场份额。

在微软公司早期的智能手机阵营中，一般分为Pocket PC Phone和Smart Phone两大系列。其中Pocket PC Phone多以手写触摸方式进行相应的操作，而Smart Phone则带有类似传统手机的数字按键。

微软推出Pocket PC、Pocket PC Phone和Smart Phone的初衷是希望在掌上电脑领域占有一席之地，希望有更多的消费者使用安装有Windows CE操作系统的掌上电脑。但是早期的Windows CE 1.0版本不支持手写识别，必须依靠键盘才可以输入，操作比较烦琐，不适合掌上电脑的使用环境。因此在与Palm公司早期的较量中，微软并不占上风。为了改变这种局面，微软公司在2000年，对Windows CE 3.0以后的版本进行了细分，其中就把针对掌上电脑的操作系统更名为Pocket PC。按照微软公司对Pocket PC的定义，“Pocket PC是一种手持设备，可以帮助您存储并检索电子邮件、联系人和约会信息，播放多媒体文件，玩电子游戏，借助MSN Messenger交换文本消息，浏览Web内容等，您将能够与台式机交换信息和同步”。在2003年，微软又推出了Pocket PC 2003平台，进一步增强了Pocket PC的功能。同时为了配合微软的.Net战略，在Pocket PC 2003 Second

Edition中，已经把.Net Framework精简版移植到了Pocket PC 2003系统的内部。当在使用Visual Studio.Net中编译、执行智能设备程序的时候就会发现，如果模拟器是Pocket PC 2003 Second Edition版本，就会省去下载.Net Framework的过程。

尽管Pocket PC可以很好地实现多媒体播放、Word、Excel文件浏览、图片查看等类似桌面电脑的功能，但是由于其缺少了对语音通话的支持，因此其市场普及程度一直不尽如人意。在这种情况下，微软又推出了基于Pocket PC架构的智能手机系统，即Pocket PC Phone。这种智能手机的尺寸和Pocket PC完全相同，两者之间最大的区别就在于Pocket PC Phone插入手机SIM卡后可以当作手机使用。在保留Pocket PC全部功能的同时，还可以通过GPRS无线访问互联网，浏览新闻，收发电子邮件。

而Smart Phone则整合了PDA和手机上网功能，特别是具有大屏幕操作的支持，把收发E-mail和记事本功能集成到了一个数据模块上。我们完全可以单手操作Smart Phone，调用其电话拨号与数据计算功能。Smart Phone这一种新型的智能电话平台把与手机操作、PDA操作相关的功能整合到一个平台上，为用户提供了数据、语音、音频、视频、电子邮件等多种功能。

表3-2列出了Pocket、Pocket PC Phone、Smart Phone三者之间的功能对比。

表3-2　Pocket、Pocket PC Phone、Smart Phone三者之间的功能对比

特　性	Pocket PC	Pocket PC Phone	Smart Phone
高分辨率彩色屏幕	有	有	有
触摸屏	有	有	无
IE浏览器	有	有	有
语音通信	无	有	有
内置GSM、GPRS或者CDMA模块	无	有	有
Media Player	有	有	有
Pocket Word	有	有	无
Pocket Excel	有	有	无
MSN Messenger	有	有	有

图3-12和图3-13分别展示了多普达的Pocket PC Phone手机外观及其采用的Pocket PC 2003 Second Edition用户界面；图3-14和图3-15分别展示了多普达的Smart Phone手机外观及其用户界面。

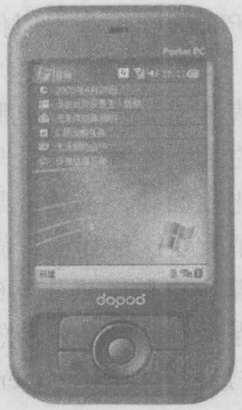


图3-12　多普达的Pocket PC Phone手机外观

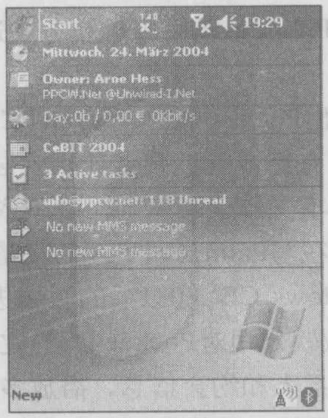


图3-13　Pocket PC 2003 Second Edition用户界面

图3-14 多普达的Smart Phone手机外观

图3-15 Smart Phone图形用户界面

伴随着微软最新桌面操作系统Vista的发布，越来越多的Windows CE爱好者也盼望着自己喜欢的Windows CE也可以尽快进行版本的更新，以便做到和Vista更好的兼容。于是微软在2006年发布了最新版本的Windows CE，即Windows Embedded CE 6.0。微软的各个合作伙伴也积极响应此次新产品的发布，纷纷推出了以Windows Embedded CE 6.0为核心的智能设备，如多普达的P860和C750。

图3-16展示了P860手机外观，图3-17展示了P860手机界面，图3-18展示了C750手机外观，图3-19展示了C750手机界面。

图3-16 P860手机外观

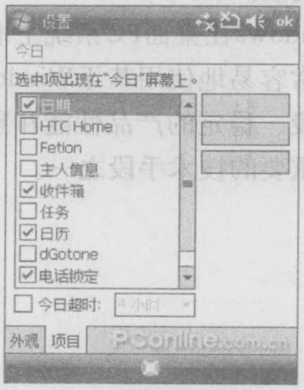


图3-17 P860手机界面

图3-18 多普达C750手机

图3-19 C750手机界面

从图3-16～图3-19中可以看到，早期Pocket PC通过手写触摸形式进行操作，Smart Phone通过键盘操作的方式正在被逐渐地改变，各个系列的设备已经出现了融合的趋势。例如，已经有越来越多的厂商推出了带有键盘的Pocket PC Phone手机，有一些高端智能设备还加入了动作感应器，即可以通过手机不同方向的移动来实现不同的控制。实际上，不管智能设备的操作形式发生怎样的变化，这一切都是以最终用户的实际需求为导向的，这实际上也是进行智能设备程序设计时所遵循的原则。

3.4.2 在嵌入式领域的应用

在这里所指的嵌入式领域一般包括仪器、工业自动化控制、医疗、信息终端等。如在工业控制领域上，就要求嵌入式操作系统有非常强的实时控制能力。Windows CE的组件化和可自定义功能非常适合工业自动化控制的要求。基于Windows CE的嵌入式产品可以满足工业、仪器、控制系统领域等稳定、经济的使用要求，它可以被应用在流程自动化系统、检测系统、远程控制系统、工业机器人等关键设备上。

而在医疗领域，针对正在开发的大量不同类型的医疗设备和系统，医疗保健行业需要一种可以自定义、可靠、稳定的操作系统作为自己产品设计的基础。Windows CE正好满足了这些要求。目前，程序开发人员已经可以利用Visual Studio.Net开发工具，开发基于Windows CE的患者监护仪、影像设备、远程和本地医疗诊断设备等嵌入式产品，在提高医院工作效率的同时，极大地方便了患者和医生。

由于Windows在桌面PC系统上取得的巨大的成就，多数人都熟悉其良好的操作界面，因此操作人员也非常容易地使用基于Windows CE的嵌入式产品。Windows CE正在凭借微软公司强大的市场营销手段、稳定的产品性能和独具特色的嵌入式产品，深入到了工业控制领域，已经成为工业控制领域重要的技术手段之一。

第4章 Windows Embedded CE 6.0开发基础

4.1 Windows Embedded CE 6.0 的开发环境

在正式进行Windows Embedded CE 6.0程序设计开发以前，需要首先熟悉一下程序的开发环境。同经常使用的桌面PC集成开发环境不同，设计Windows Embedded CE 6.0下的程序需要专业的开发工具及相应的硬件平台。在Windows CE 5.0以前，如果使用Embedded Visual C++进行相应程序的设计，需要以下一些开发工具：

- Embedded Visual C++ / Embedded Visual Basic。
- Platform Builder。
- ActiveSync。
- Windows CE SDK。

其中Embedded Visual C++或者Embedded Visual Basic提供了程序代码编写的开发环境；使用Platform Builder可以根据用户的需求定制具有不同特性的Windows CE平台；通过ActiveSync软件，可以实现智能设备（如Pocket PC等）与PC的连接，以同步二者之间的电话薄、保存音频、视频、实现智能设备程序安装等功能，同时也可以通过该软件把在PC端开发的程序下载到智能设备上；而Windows CE SDK则为整个开发计算机提供了一套针对Windows CE特性的开发环境，当安装好相应的SDK以后，就会在Embedded Visual C++或Embedded Visual Basic中看到Windows CE相关的选项，这样就可以直接创建基于Windows CE的工程文件。

硬件设备普及程度的不断提高，设备在消费者手里的生命周期正越来越短，开发者要在时间减少的同时做更多的开发，应用场景的不同导致设备的差异越来越大而我们希望在硬件和软件平台上的投资能在多个产品中用到，这些因素促使了一种更为高效的开发工具的出现，这就是Visual Studio 2005。

同采用Embedded Visual C++进行智能设备程序设计不同，在新推出的Visual Studio 2005中，Platform Builder已经作为它的一个组件运行其中。可以在微软公司的网站上免费下载具有180天使用期限的Platform Builder for CE 6.0版本。当安装好这个模块以后，就可以在Visual Studio 2005中直接进行相应平台的定制。使用这种开发方式，需要下面一些开发工具：

- Visual Studio 2005。
- ActiveSync。
- Windows Embedded CE 6.0。

上面提到的Windows Embedded CE 6.0安装包中包含了Platform Builder for CE 6.0及相关的开发环境。

图4-1展示了安装好Platform Builder for CE 6.0后的Visual Studio 2005集成开发环境。

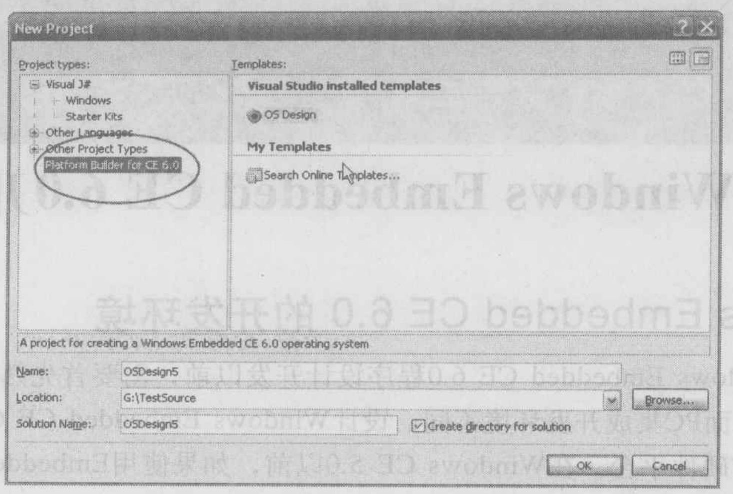


图4-1 Platform Builder for CE 6.0在Visual Studio 2005中的显示

4.2 Visual Studio 2005简介

4.2.1 Visual Studio 2005的特点

作为微软曾经提出的.Net战略中非常重要的一个环节，无论是早期的Visual Studio.Net 2002、Visual Studio.Net 2003，还是到最新的Visual Studio 2005，它们都具备了.Net框架的一些特点。

表4-1列出了.Net框架相对于其他平台的一些特点。

表4-1 .Net平台特点

特　点	描　述
互操作支持	可以使我们比较容易地从其他平台或者编程语言过渡到该平台上
公共语言运行时（CLR）	.Net平台所有语言共享的引擎。它包括C#、VB.Net等所支持的语言
基类库（BCL）	如果学习过Java，就会知道Java之所以受欢迎，除了它的托管执行环境和跨平台的支持以外，另外一个非常重要的因素就是它的类库。.Net平台也接受了类库的概念，并且还进一步提升了跨语言的支持。现在BCL支持的特性，如远程处理、字符串处理、异常处理等任何遵守CLI的编程语言都是一样的
XML核心内容	程序化、流处理、解析、转换和架构支持已经成为.Net运行时中的特性
简洁的部署	在.Net平台下，可以很方便地来部署程序集、ASP.Net文件、配置文件等
面向对象的ASP.Net	灵活地适应现有的OO框架，利用经过编译的服务器端代码增强Web应用程序的性能

在这里用到的Visual Studio是微软公司出品的一款大型应用软件，从最初的Visual Studio 97开始就成为编程的重要工具。Visual Studio的开发经历了Visual Studio 97、Visual Studio 6.0、Visual Studio.NET（2002）、Visual Studio.NET 2003、Visual Studio 2005，直到2008年刚刚发布的Visual Studio 2008。

在这里需要提及的是，进行.NET开发的工具并非只有Visual Studio，实际上最常见的如Windows自带的记事本实用程序也可以实现.NET程序开发。但是通常情况下Visual Studio已经成

为了.NET开发的首选工具。事实上，除了Visual Studio和记事本之外，还有其他的开发工具可供选择。尽管这些工具没有成为本书的首选，读者还是可以选择性地关注一下这些工具的进展。这些开发工具包括Sharp Develop、Borland C# Builder和Mono Develop。此外，还有许多程序员喜爱的编辑器可供使用，如Editplus和UltraEdit等。

本书中介绍使用的Visual Studio 2005是一套完整的开发工具集，包含了大量的功能。它主要用于生成ASP.NET Web应用程序、XML Web Services、桌面应用程序和移动应用程序。Visual Basic、Visual C++、Visual C# 和Visual J# 全都使用相同的集成开发环境（IDE）。利用此IDE可以共享工具且有助于创建混合语言解决方案，使程序可以使用不同的语言共同开发。另外，这些语言利用了.NET Framework的功能，通过此框架可使用简化ASP Web应用程序和XML Web Services开发的关键技术。

Visual Studio 2005的集成开发环境中还为开发人员提供了大量的实用工具以提高工作效率。这些工具包括自动编译、项目创建向导、创建部署工程等。Visual Studio 2005还包括许多非常实用的功能，希望读者在不断的使用中掌握这些工具。

图4-2展示了Visual Studio 2005的图形用户界面。

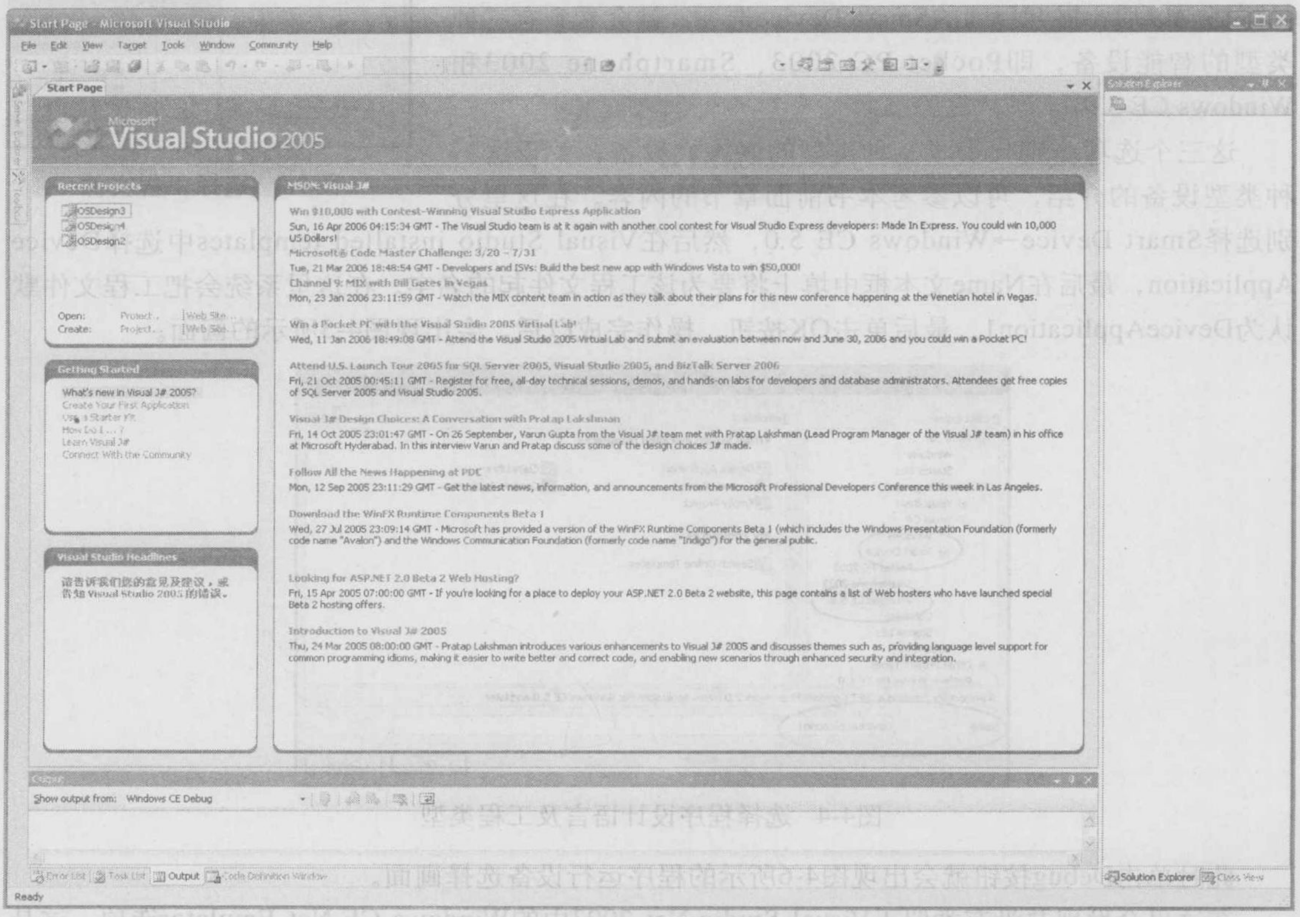


图4-2 Visual Studio 2005的图形用户界面

与Visual Studio 2003相比，Visual Studio 2005不仅仅是在.NET版本的支持上发生了变化，其在许多方面都提供了改进乃至全新的功能。

Visual Studio 2005的新增功能集中在以下几个方面：

- 开发环境中的新增功能。
- 代码编辑中的新增功能。
- 项目、解决方案中的新增功能。
- 生成、测试和部署中的新增功能。
- Visual Studio 2005调试器的新增功能。
- Visual Studio 2005中扩展性和自动化的新增功能。
- Visual Studio帮助中的新增功能。

4.2.2 使用Visual Studio 2005构建Smart Device工程

首先启动Visual Studio 2005，选择File→New→Project，此过程如图4-3所示。

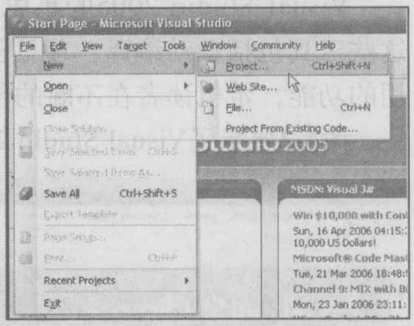


图4-3 选择工程文件

在出现的图4-4中，可以选择一种程序设计语言，如Visual Basic或者Visual C#，（在这里以Visual C#为例加以说明）。当选择好语言以后，展开里面的Smart Device选项，就会看到有三种类型的智能设备，即Pocket PC 2003、Smartphone 2003和Windows CE 5.0。

这三个选项分别代表了三种类型的嵌入式设备，关于这三种类型设备的介绍，可以参考本书前面章节的内容。在这里分别选择Smart Device→Windows CE 5.0，然后在Visual Studio installed templates中选择Device Application，最后在Name文本框中填上将要为该工程文件起的名字。在这里系统会把工程文件默认为DeviceApplication1。最后单击OK按钮。操作完成以后，会出现图4-5所示的画面。

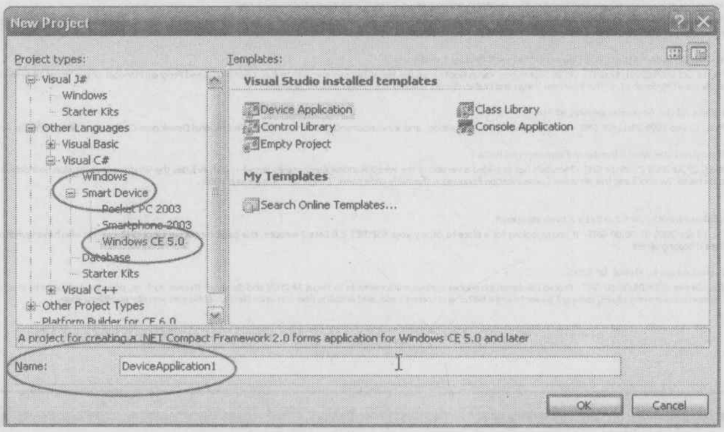


图4-4 选择程序设计语言及工程类型

此时单击Debug按钮就会出现图4-6所示的程序运行设备选择画面。

在这里会发现并没有类似于Visual Studio.Net 2003中的Windows CE.Net Emulator选项。这是因为在Visual Studio 2005中开发环境并没有安装默认的Windows CE模拟器，如果需要把程序下载到模拟器，则需要使用Platform Builder for CE 6.0自行定义一个模拟器的SDK，然后再安装这个SDK。安装成功以后，就会在图4-7所示的对话框中看到相应的Windows CE模拟器。

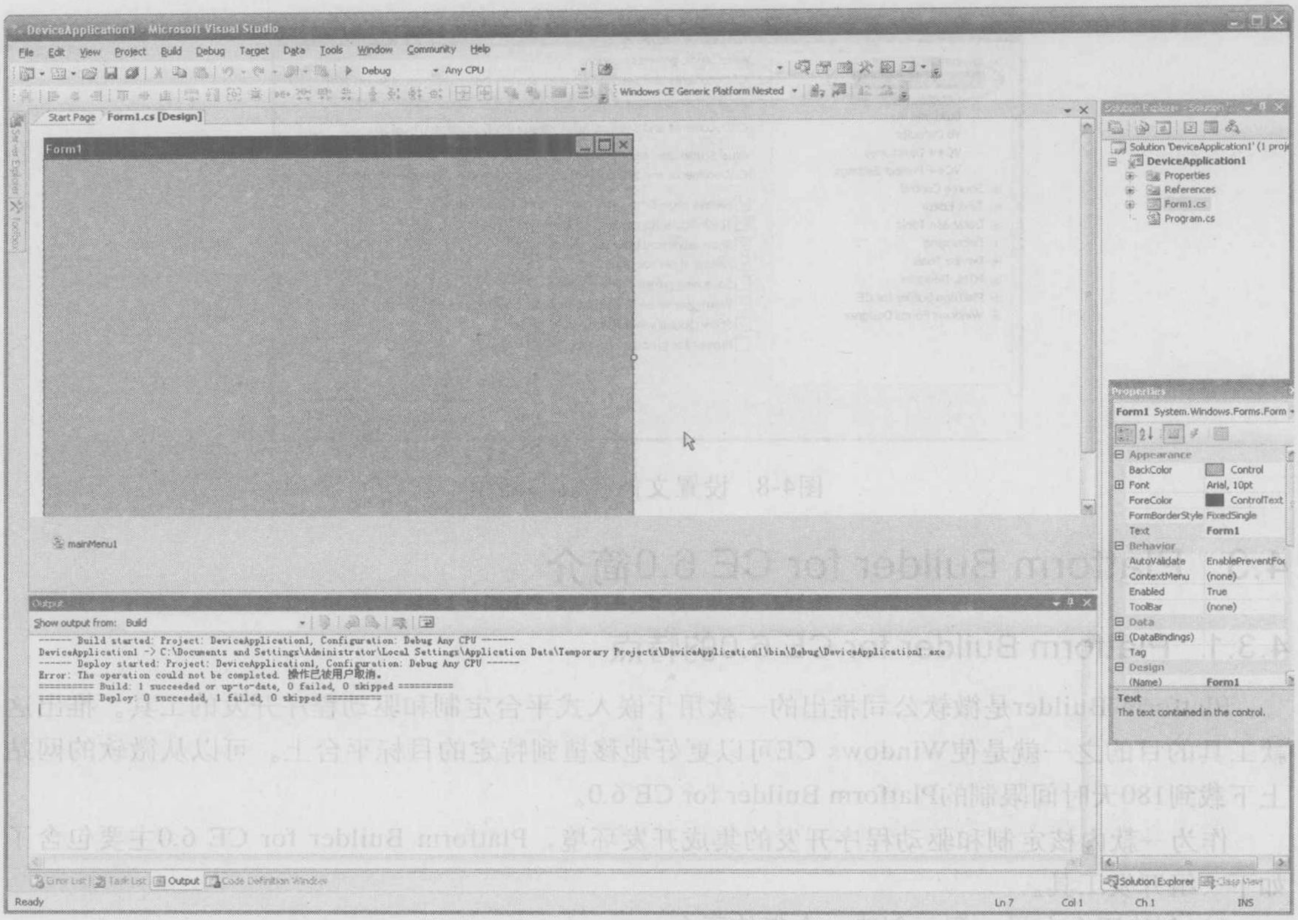


图4-5 Windows CE程序设计界面

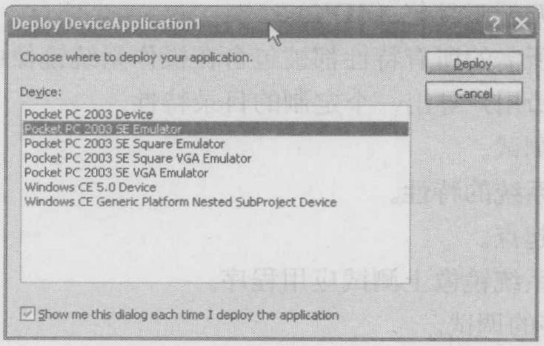


图4-6 程序运行设备选择画面

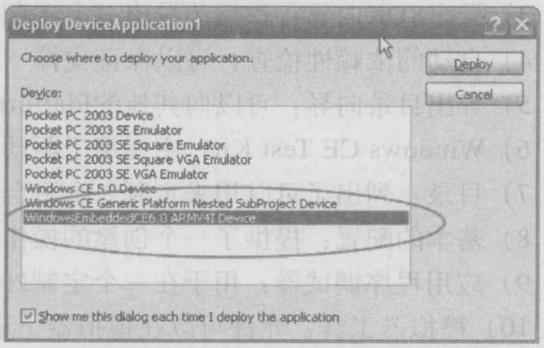


图4-7 安装SDK以后的设备部署选项

如果使用过Visual Studio.Net 2003，就会发现在VS2003中选择工程类型的那一步可以设置工程存在的位置，那么在Visual Studio 2005中又该如何设置文件的存放路径呢？

启动Visual Studio 2005后，单击Tools→Options，则会出现图4-8所示的画面。

在图4-8中选择Projects and Solutions，在右侧选项框的Visual Studio projects location中就可以设置工程文件的存放路径。

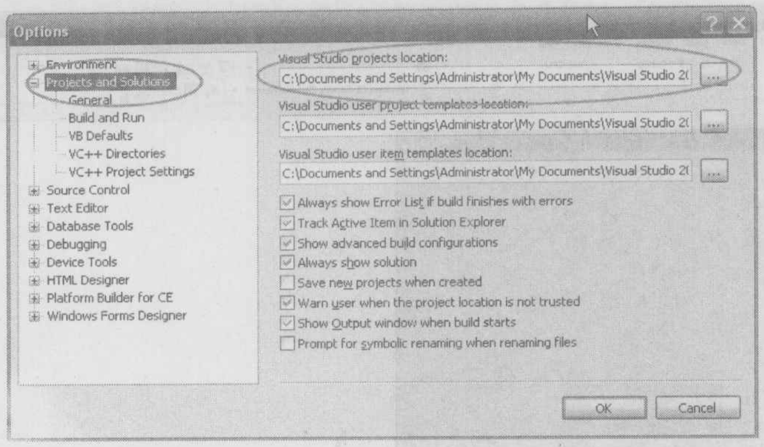


图4-8 设置文件存放的路径

4.3 Platform Builder for CE 6.0简介

4.3.1 Platform Builder for CE 6.0的特点

Platform Builder是微软公司推出的一款用于嵌入式平台定制和驱动程序开发的工具。推出这款工具的目的之一就是使Windows CE可以更好地移植到特定的目标平台上。可以从微软的网站上下载到180天时间限制的Platform Builder for CE 6.0。

作为一款内核定制和驱动程序开发的集成开发环境，Platform Builder for CE 6.0主要包含了如下一组开发工具。

1）创建平台向导：用于创建一个新的平台。

2）创建BSP向导：用于创建一个新的BSP（板级支持包）。

3）导出SDK向导：作用为导出一个定制的软件开发工具包（SDK）。

4）自动的依赖性检查：可以保证支撑一个平台所需的所有特性都被包含在操作系统镜像中。

5）导出目录向导：可以向其他的Platform Builder用户导出一个定制的目录特性。

6）Windows CE Test Kit：主要用于驱动程序的测试。

7）目录：列出了可以用来定制一个平台的操作系统的特性。

8）基本的配置：提供了一个创建的操作系统的起点。

9）应用程序调试器：用于在一个定制好的操作系统镜像上测试应用程序。

10）模拟器工具：允许可以在模拟器上进行程序的调试。

11）远程调试工具：用于在目标机上进行程序的调试。

12）内核调试器：用于调试一个定制的操作系统镜像。

我们知道，一个完整的操作系统镜像是基于几个固定的硬件平台来运行的，因此一个Windows CE系统通常情况下会包含下面的一些基本功能。

- BootLoader：BootLoader的作用是加载Windows CE嵌入式操作系统的镜像。
- CPU初始化代码：它是基于特定CPU的初始化代码，由Windows CE提供。这里的特定CPU指的是我们定制的嵌入式操作系统是基于X86系列，还是MIPS系列等。根据CPU类型的不同，其初始化代码也就不同。

- 相应的驱动程序：这里指的驱动程序通常包括嵌入式操作系统的键盘、鼠标、声卡、显卡、USB驱动程序等。
- 用户界面接口。
- 完成特殊功能的一些应用程序。

同时，Platform Builder for CE 6.0还具备了微软其余软件的特性，如类似的操作界面（通用的按钮图标、通用的快捷键配置等）、友好的开发环境等。

图4-9展示了Platform Builder for CE 6.0的开发环境。

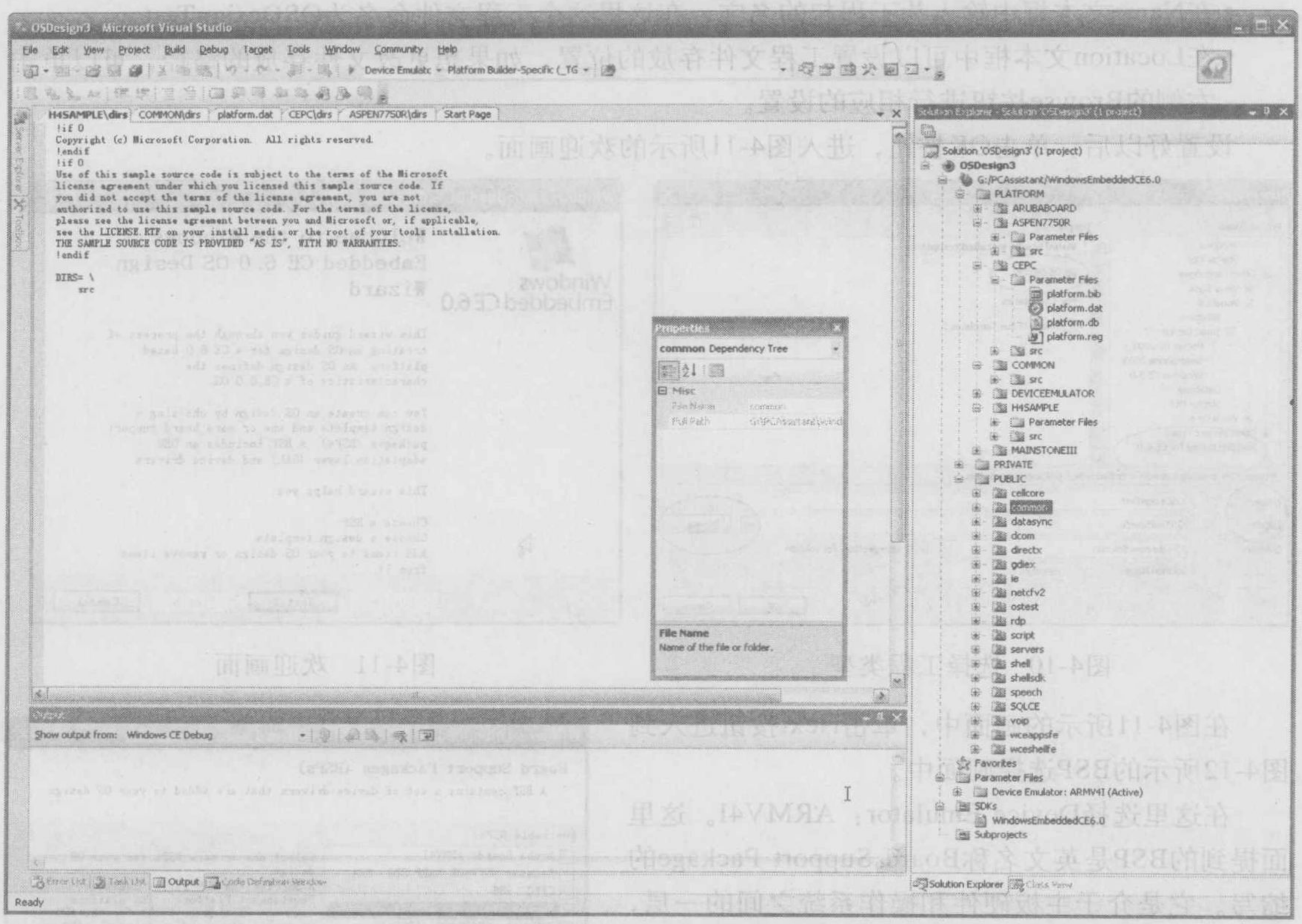


图4-9 Platform Builder for CE 6.0的开发环境

4.3.2 使用Platform Builder for CE 6.0进行平台的定制

使用Platform Builder for CE 6.0定制一个平台，主要包括下面几个部分：

1）生成操作系统镜像。

2）生成BSP包，主要包括OAL（OEM Adaptive Layer）和驱动程序。

3）添加新的组件，主要包括驱动程序和应用程序。

4）修改平台的组件。

5）设备平台的属性和环境变量。

6）设置远程连接属性，以便进行操作系统的下载。

7）下载操作系统，主要包括下载操作系统到硬件平台和下载操作系统到模拟器。

8）进行相应的调试，主要包括硬件平台的调试和模拟器的调试。

下面就来看一下如何使用Platform Builder for CE 6.0来定制一个基于Windows CE 6.0的新平台。

首先启动Visual Studio 2005，单击File→New→Project，此时会出现图4-10所示的选择工程类型画面。然后进行下面的设置：

- 在Project types中选择Platform Builder for CE 6.0；
- 在Visual Studio installed templates中选择OS Design；
- 在Name文本框中输入为工程起的名字，在这里这个工程文件命名为OSDesignTest；
- 在Location文本框中可以设置工程文件存放的位置，如果想更改文件存放的路径，可以单击右侧的Browse按钮进行相应的设置。

设置好以后，单击OK按钮，进入图4-11所示的欢迎画面。

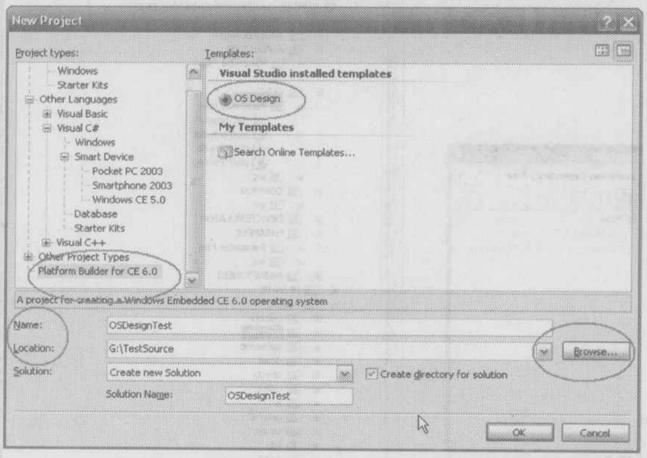


图4-10 选择工程类型

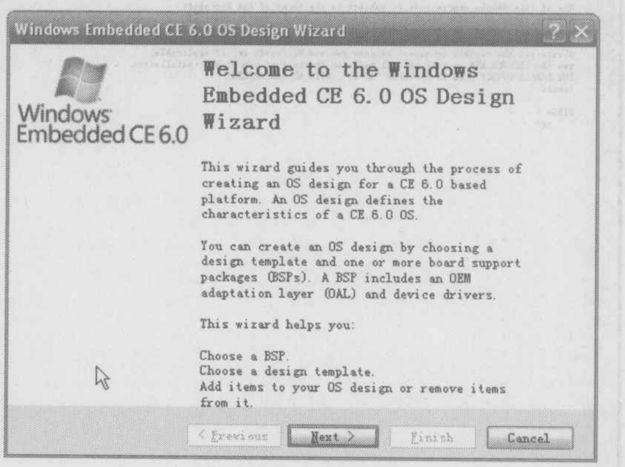


图4-11 欢迎画面

在图4-11所示的画面中，单击Next按钮进入到图4-12所示的BSP选择画面中。

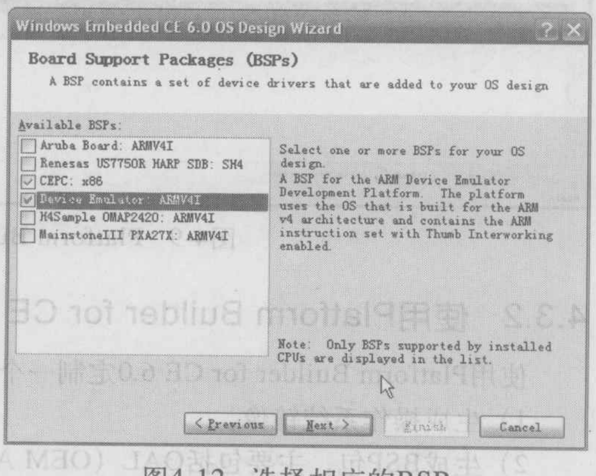


图4-12 选择相应的BSP

在这里选择Device Emulator：ARMV4I。这里面提到的BSP是英文名称Board Support Package的缩写。它是介于主板硬件和操作系统之间的一层，也可以认为是属于操作系统的一部分。其主要作用是支持操作系统，使之可以更好地适应计算机的主板。一般地，BSP是相对于操作系统而言的，因此不同的操作系统对应于不同定义形式的BSP。例如，Windows CE的BSP和Linux的BSP相对于某一CPU来说尽管实现的功能一样，可是写法和接口定义则完全不同。纯粹的BSP所包含的内容一般来说是和系统相关的驱动或应用程序，如网络驱动程序、网络协议、串口驱动程序和下载调试等，离开这些驱动系统就不能正常工作了。

表4-2列出了Windows Embedded CE 6.0所支持的BSP。

表4-2 Windows Embedded CE 6.0所支持的BSP

系列	BSP	Kernel	是否包含源代码
ARM	Intel Mainstone III(C-Step)	ARMv4i	Yes
	Device Emulator	ARMv4i	Yes
	Board	ARMv4i	Yes
	TI OMAP 2420(ARM v6 core)	ARMv4i	Yes
MIPS	NEC Rockhopper SG2 Vr5500	MIPSII&II_FP,MIPSSIV$IV_FP	Yes
SH4	Renesas Aspen	SH4	Yes
X86	X86(CEPC)	X86	Yes

完成BSP选择以后，单击Next按钮，进入到图4-13所示的设置模板选项。

在这里选择PDA Device，然后单击Next进入到图4-14所示的模板变量选择界面。在这里选择Mobile Handheld，然后单击Next按钮进入下一个界面。

在如图4-15所示的程序和多媒体选项中，可以根据所定制的模拟器的需要来进行相应的选择。该步骤中各项的含义如下。

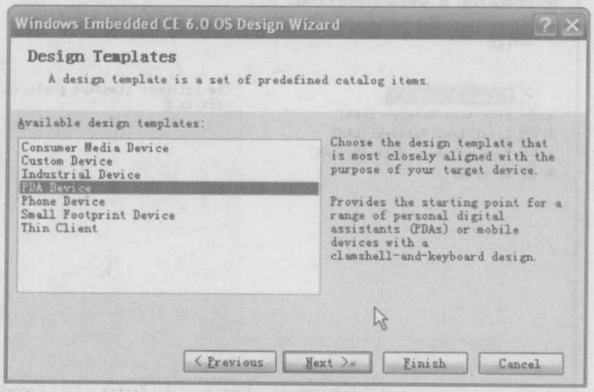


图4-13 模板的选择

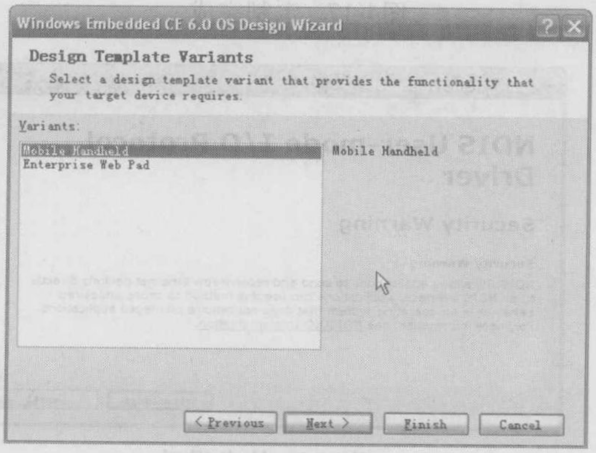


图4-14 模板变量选择

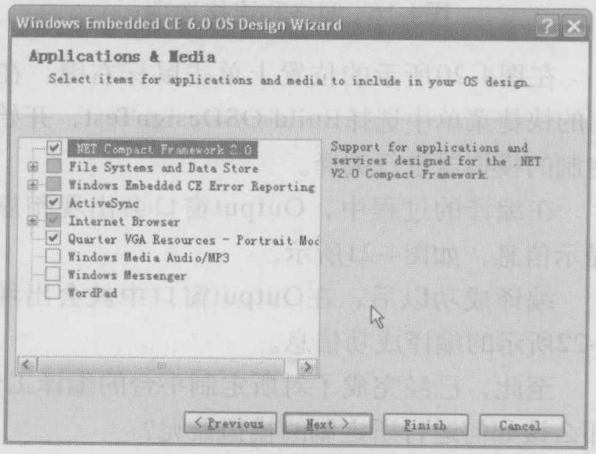


图4-15 程序和多媒体选择

- .NET Compact Framework 2.0：包含.NET Compact Framework 2.0框架。
- File System and Data Store：包含文件系统和数据存储。
- Windows Embedded CE Error Reporting：包含Windows Embedded CE错误报告。
- ActiveSync：包含ActiveSync，用于PC与移动设备的连接。
- Internet Brower：包含IE浏览器。
- Quarter VGA Resources-Portrait Mode：使设备支持320×240分辨率的QVGA屏幕。
- Windows Media Audio/MP3：设备支持音频和MP3播放功能。

- Windows Messenger：包含即时通信软件Messenger。
- WordPad：包含办公软件WordPad。

根据需要选择好相应的选项以后，单击Next按钮进入到图4-16所示的网络和连接选项。同样在这里根据实际的需要进行相应内容的选择。然后单击Next按钮，进入下一个步骤。

此时会出现图4-17所示的定制完成提示界面。在这个界面中，单击Finish按钮，会出现图4-18所示的信息提示界面。不用去管它，单击Acknowledge按钮完成相应内容的选择。此时会出现图4-19所示的Platform Builder for CE 6.0图形用户界面。

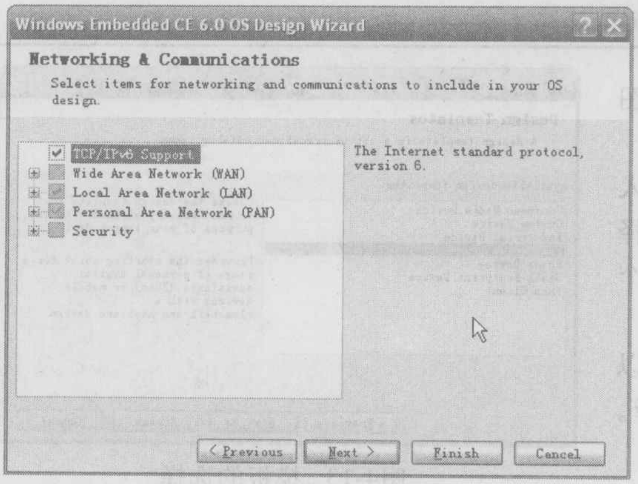


图4-16　网络和连接选项

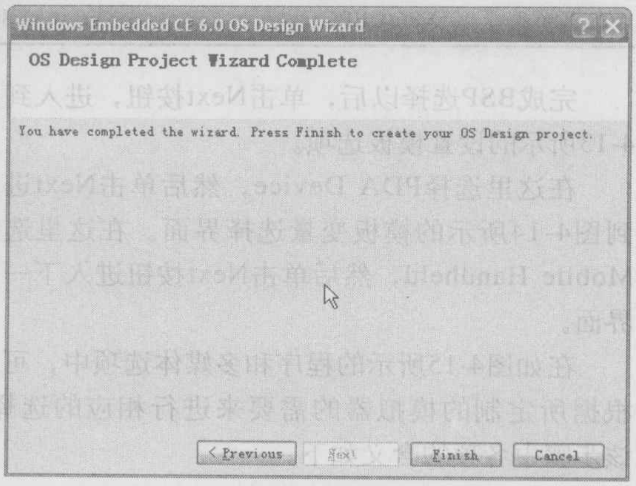


图4-17　定制完成

在图4-20所示的位置上单击鼠标右键，在弹出的快捷菜单中选择Build OSDesignTest，开始对定制的模拟器进行编译。

在编译的过程中，Output窗口会出现相应的提示信息，如图4-21所示。

编译成功以后，在Output窗口中就会出现图4-22所示的编译成功信息。

至此，已经完成了对所定制平台的编译工作，那么该如何运行所定制的模拟器呢？

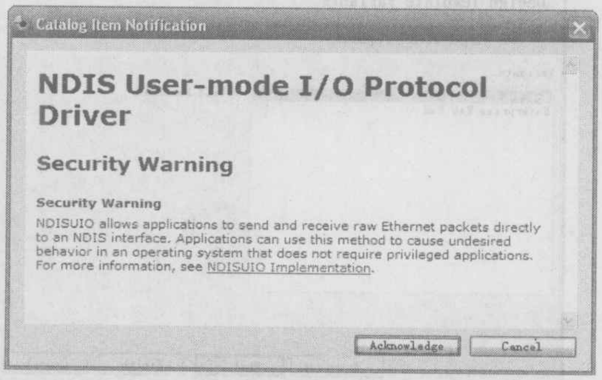


图4-18　信息提示

在运行定制的模拟器之前，需要对模拟器进行相应的参数设置。首先单击Target，然后选择Connectivity Options，此时会出现图4-23所示的画面。

在图4-23所示的对话框中，需要进行下面的设置：

- Download：选择Device Emulator（DMA）。
- Transport：选择Device Emulator（DMA）。
- Debugger：选择KdStub。

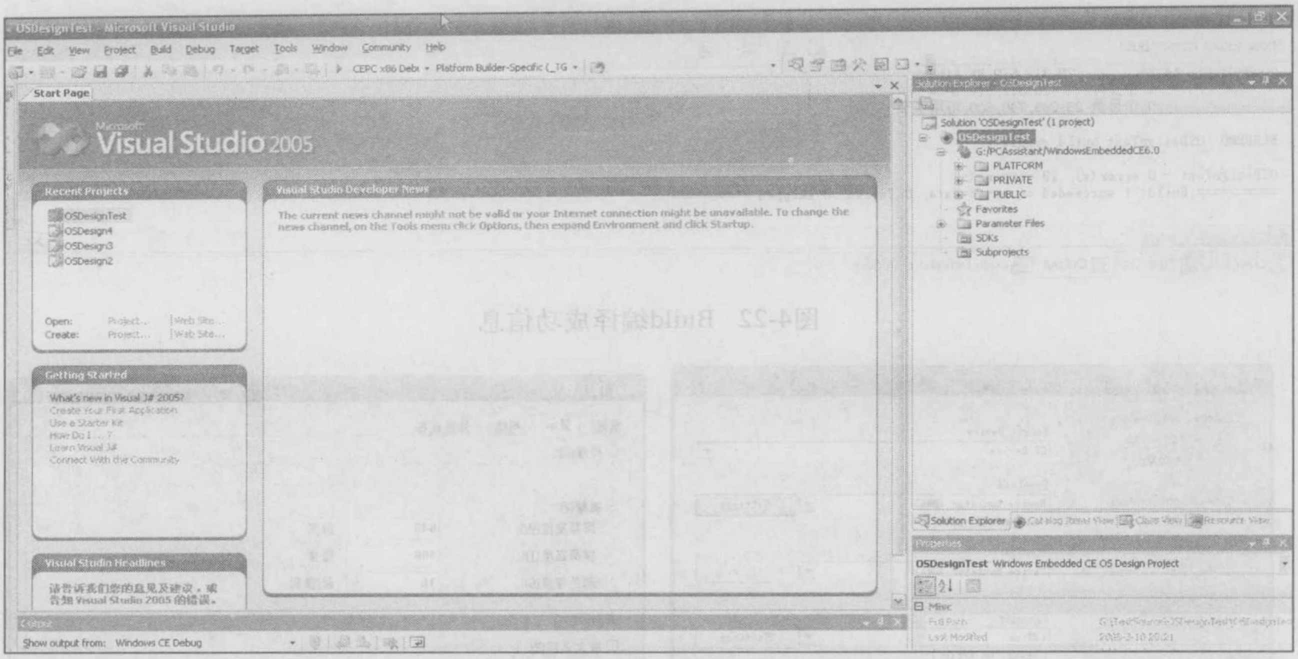


图4-19 Platform Builder for CE 6.0图形用户界面

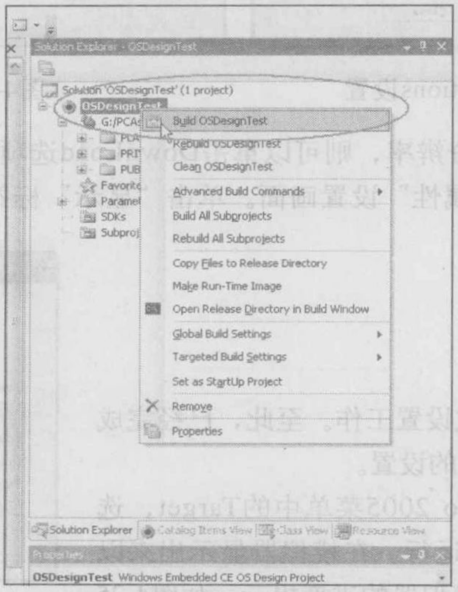


图4-20 Build OSDesignTest

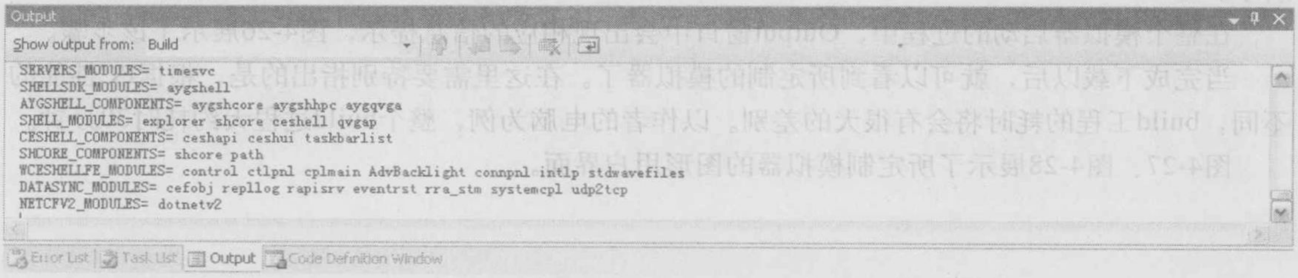


图4-21 编译过程中相应的提示信息

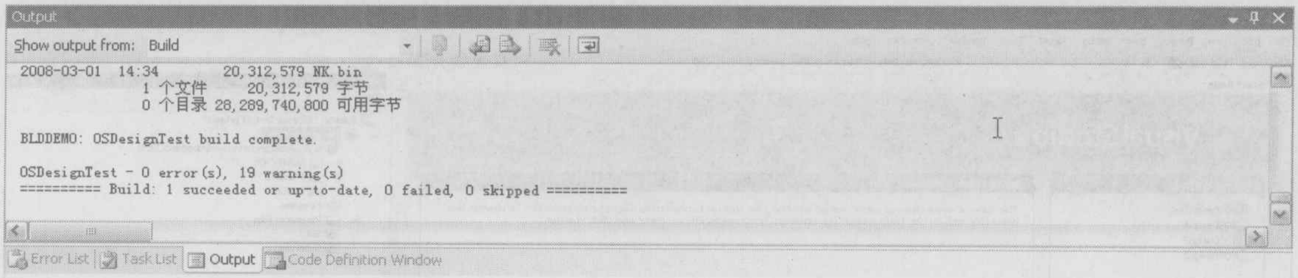


图4-22 Build编译成功信息

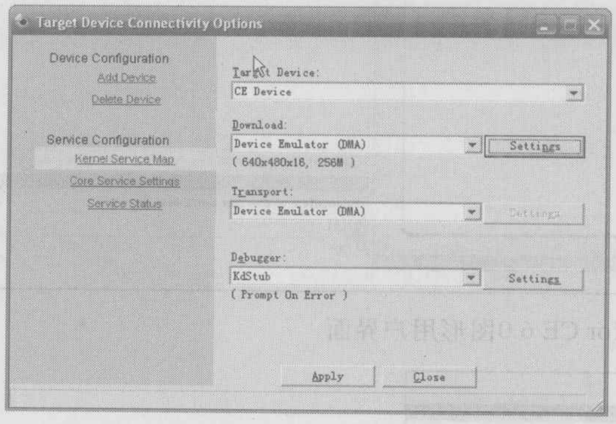


图4-23 Connectivity Options设置

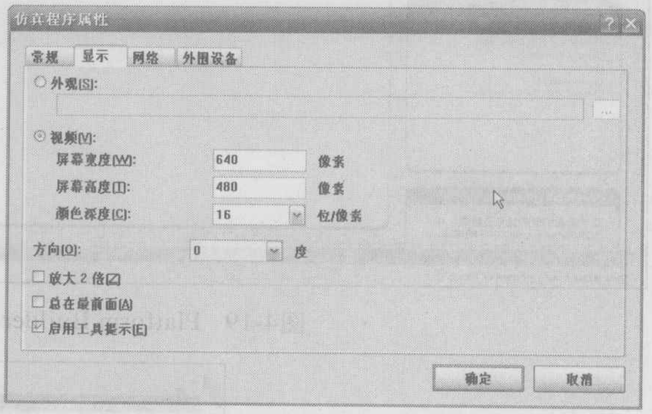


图4-24 仿真程序属性

如果想设置模拟器屏幕的分辨率，则可以单击DownLoad选项右侧的Settings按钮。此时会出现如图4-24所示的“仿真程序属性”设置画面。单击“显示”标签，然后选择“视频（V）”，对里面的选项进行如下设置：

- 屏幕宽度：640。
- 屏幕高度：480。
- 颜色深度：16。

然后单击“确定”按钮完成设置工作。至此，已经完成了对所定制的模拟器的显示属性的设置。

接下来，单击Visual Studio 2005菜单中的Target，选择Attach Device进行模拟器的运行。在模拟器显示出来以前，首先系统会展示一个下载模拟器的进度提示，如图4-25所示。

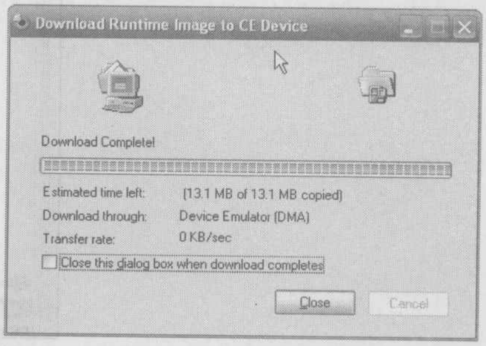


图4-25 下载进度提示

在整个模拟器启动的过程中，Output窗口中会出现相应的信息提示，图4-26展示了该步骤。

当完成下载以后，就可以看到所定制的模拟器了。在这里需要特别指出的是，根据PC性能的不同，build工程的耗时将会有很大的差别。以作者的电脑为例，整个build过程大约耗时30分钟。

图4-27、图4-28展示了所定制模拟器的图形用户界面。

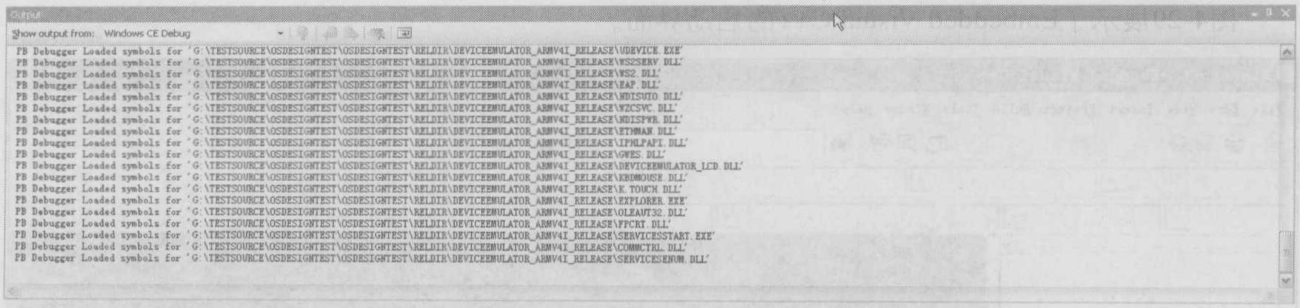


图4-26 模拟器启动过程中的信息提示

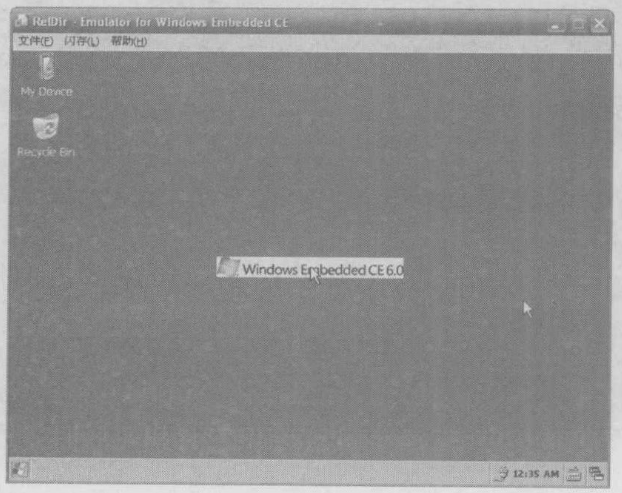


图4-27 所定制模拟器的图形用户界面1

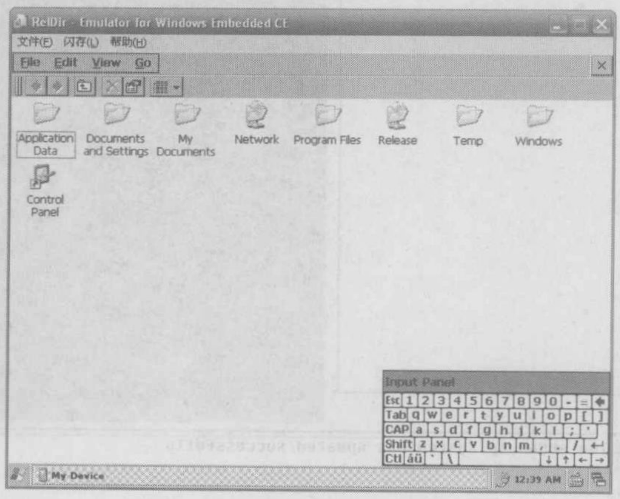


图4-28 所定制模拟器的图形用户界面2

4.4 Embedded Visual C++

4.4.1 Embedded Visual C++简介

从前面的介绍中可以看到，目前很多的程序开发人员都采用Visual Studio作为Windows CE程序设计的开发工具。但是在一些公司中，仍旧有部分程序员使用Embedded Visual C++作为开发工具。由于本书的重点是介绍Visual Studio 2005，因此在这里只简单介绍一下Embedded Visual C++的特点。

Embedded Visual C++主要具有以下组件：

1）包含C和C++的编译器。

2）能够管理不同平台的平台管理器。

3）Embedded Visual C++下包含一个模拟器，可以在PC下模拟Windows CE的物理平台。

4）包含类视图、资源视图、文件视图等功能的工作区。

5）用于远程调试的工具。

6）能够单步执行程序和设置断点的调试器。

7）能够创建类和应用程序的类向导和应用程序向导。

8）相应的帮助文档。

图4-29展示了Embedded Visual C++的启动界面。

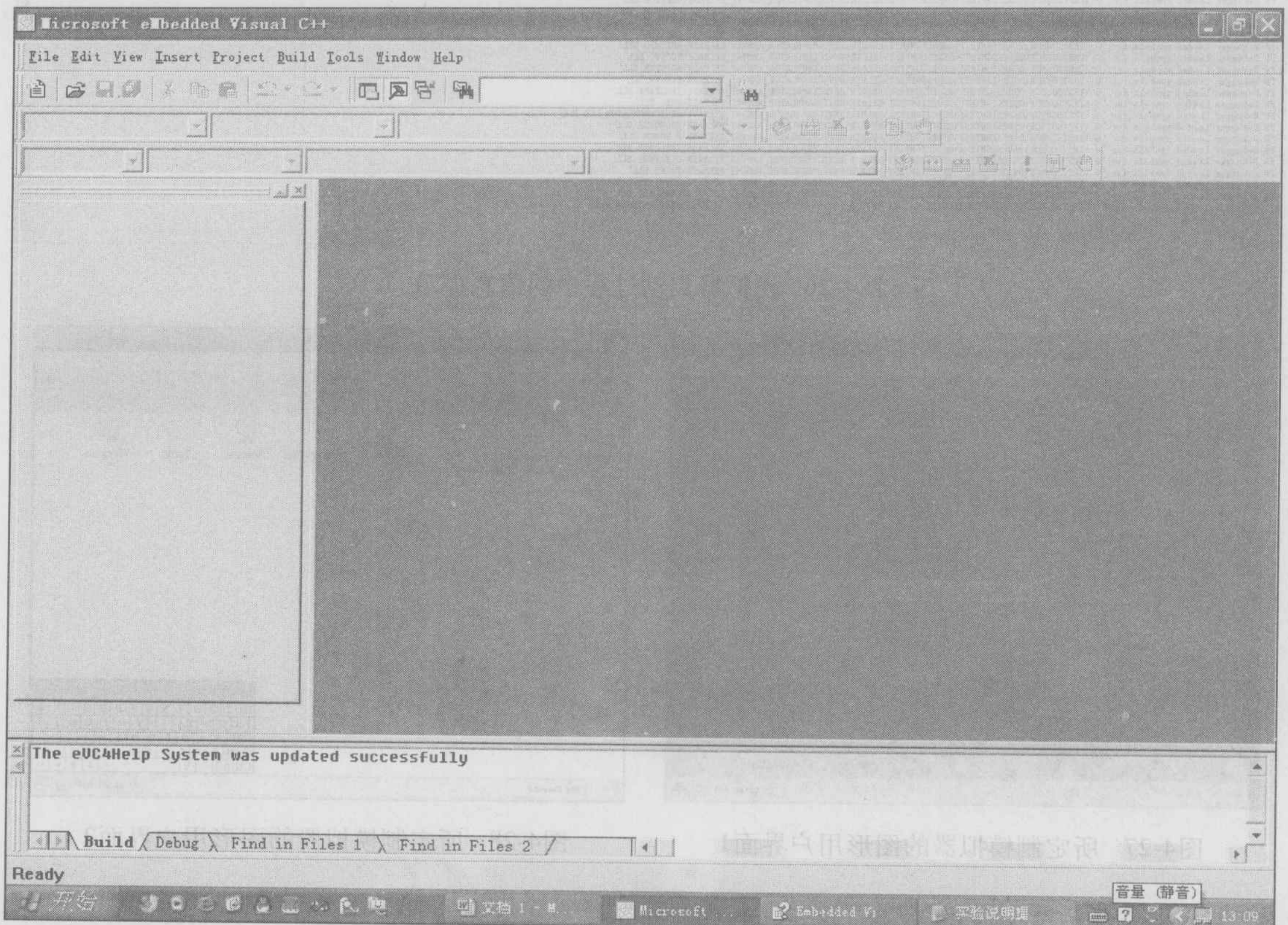


图4-29 Embedded Visual C++的启动界面

下面看一下EVC中的工具栏和菜单。工具栏是EVC开发中非常重要的辅助工具，工具栏将一些常用任务集成化为工具栏上的按钮。EVC带有一个预先定义好的工具栏集，在编译、调试等各种任务中发挥了重要的作用，在Tools菜单下选择Customize就可以自定义工具栏，该工具栏如图4-30所示。

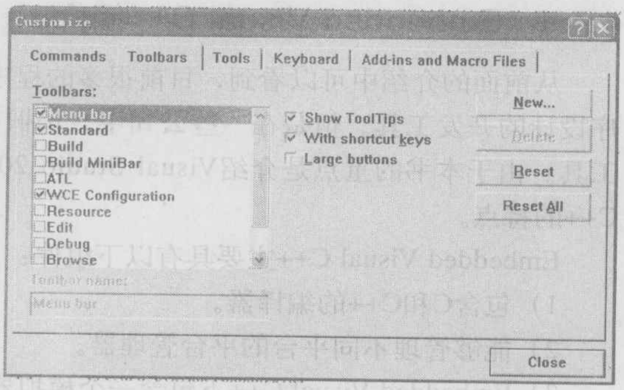


图4-30 EVC中的自定义工具栏

用户可以设置使用哪些工具栏及工具栏的风格（显示工具栏提示，使用快捷键，大图标显示），还可以建立自己的工具栏。图4-31是一个工具栏以浮动窗口显示的例子。

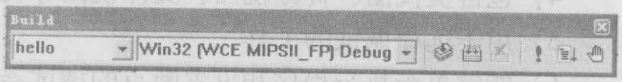


图4-31 以浮动窗口显示的工具栏

工具栏可以作为一个单独的窗口显示，同时，每个工具栏又都是可停靠的。工具栏处于停靠状态时位于菜单的下面，可以独占一行，也可共占一行。每个工具栏都有自己的手柄，拖动手柄

便可移动其位置，或使其在停靠状态和浮动状态之间切换。停靠状态的工具栏如图4-32所示。

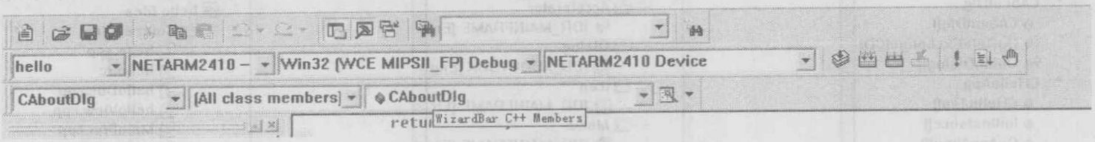


图4-32 停靠状态的工具栏

单击工具栏区的空白处弹出工具栏菜单，在菜单中也可以定制工具栏，此步骤如图4-33所示。可以把EVC的菜单栏看做是一种特殊的工具栏。菜单栏一般位于工具栏的上方，最左端是手柄，这意味着菜单栏可以同工具栏一样被拖拉，但是，与工具栏不同的是，除去全屏模式，菜单栏总是可见的。

进入全屏模式可以选择View菜单下的Full Screen选项，如图4-34所示。

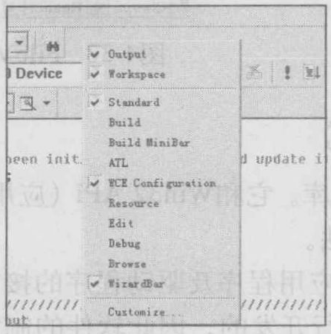


图4-33 工具栏菜单

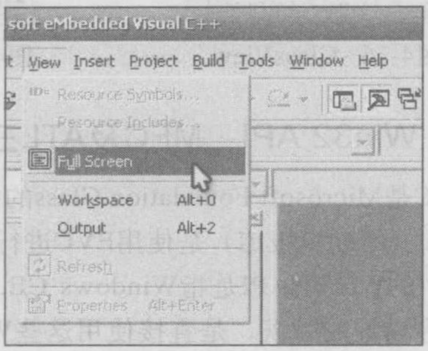


图4-34 进入到全屏显示模式

如果想了解更多有关菜单项中命令的使用，可参考EVC帮助文档。

在Embedded Visual C++中，编辑调试应用程序时，最重要的两个窗口是Workspace（工作区）窗口和Output（输出）窗口。其中的输出窗口用于显示运行编译调试时的一些系统信息，如程序异常，编译通过等。图4-35为刚刚启动编辑环境时的输出窗口。

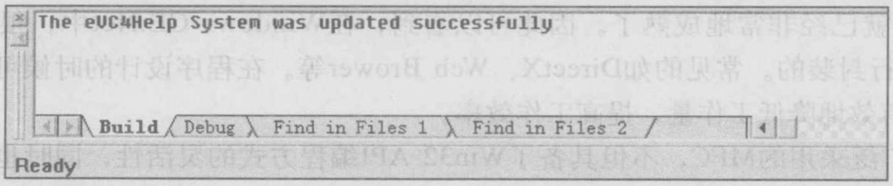


图4-35 输出窗口

在输出窗口底部有四个选项卡，分别为Build、Debug、Find in Files 1、Find in Files 2，前两个选项卡最为重要，Build选项卡显示编译器、链接器和其他工具的状态消息，Debug选项卡用于显示来自调试器的状态消息，一般包括程序异常、内存异常等。

而在工作区的底部，则有三个选项卡，它们分别为ClassView、ResourceView、FileView。对应的功能为：显示当前工程中类的结构，显示当前工程中资源组织结构，显示当前工程中文件组织结构。三个分类视图都以树形图显示，单击加号可展开分支，单击减号可以收起分支。

图4-36～图4-38分别展示了ClassView、ResourceView和FileView。

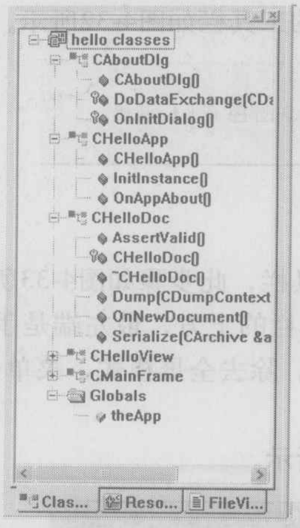


图4-36 ClassView

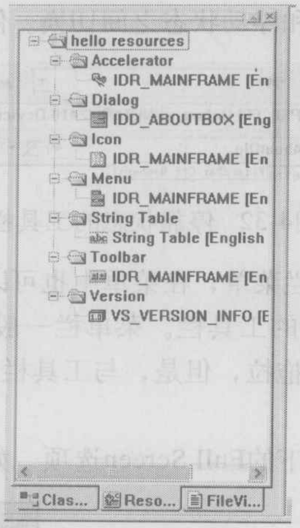


图4-37 ResourceView

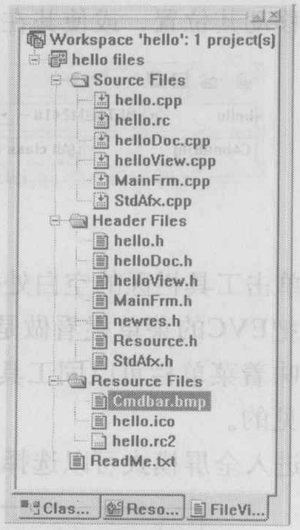


图4-38 FileView

4.4.2 Win32 API、MFC及ATL三种程序设计方式

MFC是Microsoft Foundation Class的简称，即微软基础类库。它和Win32 API（应用程序接口）以及ATL（活动模板库）是使用EVC进行程序开发的三种方法。

其中的Win32 API是指Windows CE.Net内核提供给上层应用程序及驱动程序的接口函数。在进行程序设计的时候，是直接使用这些Win32 API函数来进行开发的。因此软件的制定效率非常高，同时还可以非常灵活地控制软件的各个功能，同时保证软件可以稳定地运行。同传统的Windows API函数不同，EVC中使用的API函数只是Windows API函数的一个子集。同时即使是同名的API函数，在参数的设置上也会存在不同，如EVC中的API函数不支持安全属性，因此这些属性在EVC中都被设置成NULL。正如任何事物都有两面性一样，既然Win32 API程序设计方式非常的灵活，也就导致了在程序设计过程中的复杂程度相对较高。

前面提到的ATL就是利用COM组件技术来进行程序设计。实际上，在Windows CE.Net出现以前，COM技术就已经非常地成熟了。因此可以看到，在Windows CE.Net中，很多组件都是用COM标准来进行封装的。常见的如DirectX、Web Brower等。在程序设计的时候可以直接使用这些组件，以便有效地降低工作量，提高工作效率。

在本书中广泛采用的MFC，不但具备了Win32 API编程方式的灵活性，同时也具备了在界面编写、程序框架设计等方面的便利性。可以说，大多数使用VC开发的软件都全部或者部分地采用了MFC来进行编写。在EVC中使用MFC进行程序设计，可以通过控件拖拽的方式在界面上进行相应的布置，极大地提高了界面设计的效率。

下面就来具体地看一下如何在EVC中构建一个MFC程序。

4.4.3 在Embedded Visual C++中构建一个MFC程序

在EVC下开始编写一个应用程序，最方便的方法就是使用其内置的应用程序向导。选择File菜单下的New选项，出现如图3-39所示的界面。

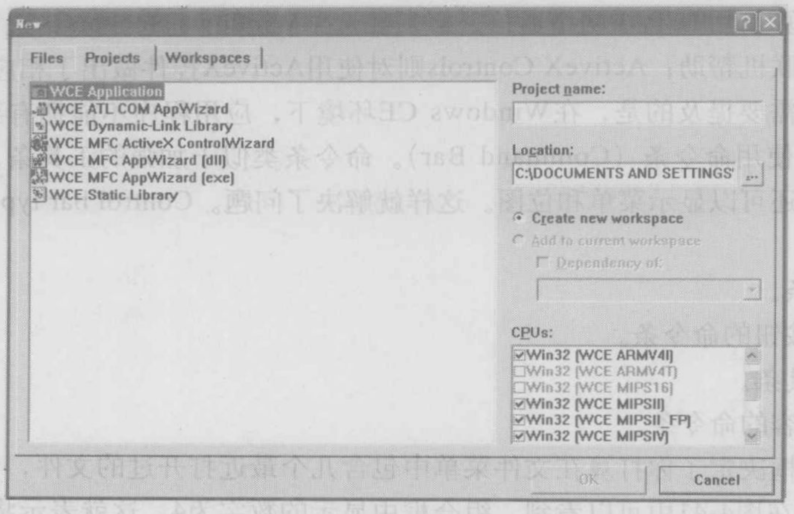


图4-39 进行工程的选择

该窗口用于创建工程，其中包括：Windows CE应用程序、Windows CE ATL COM应用程序、Windows CE动态链接库、基于MFC 的Windows CE 控件、基于MFC 的Windows CE 应用程序和Windows CE静态库等。

在右侧的Project name表示工程名，Location表示该工程存储位置，CPUs表示该工程即将运行的平台。当不打算将此工程运行于某个平台的时候，只需将前面的勾去掉即可；如果发现在CPU列表中没有目标平台，就需要去微软网站下载一个该目标平台的SDK（Software Development Kit，软件开发包），或者找到相应的产品SDK安装包，并安装好。

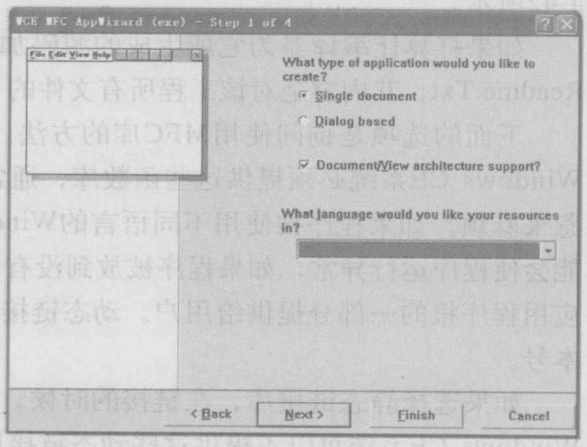


图4-40 窗口类型的选择

如果想要建立一个MFC应用程序，可以选择WCE MFC AppWizard[exe]，指定文件名和路径，单击OK按钮。然后会出现图4-40所示的窗口。在这个窗口中，将要进行窗口类型的选择。其中，Single document表示该程序是一个单文档窗口程序，Dialog based表示该程序是一个基于对话框的程序。

单文档窗口程序是指在一个主窗口中只能有一个子窗口，如Windows XP下的记事本。Windows CE不支持多文档窗口程序。基于对话框的程序模式非常适用于那些并不一定需要主窗口的小巧的实用类程序，对话框本身也是一个窗口，却比单文档程序更加灵活。

选择好以后，单击Next按钮，进入到下一个页面，进行用户特性的选择。用户特性选择界面如图4-41所示。

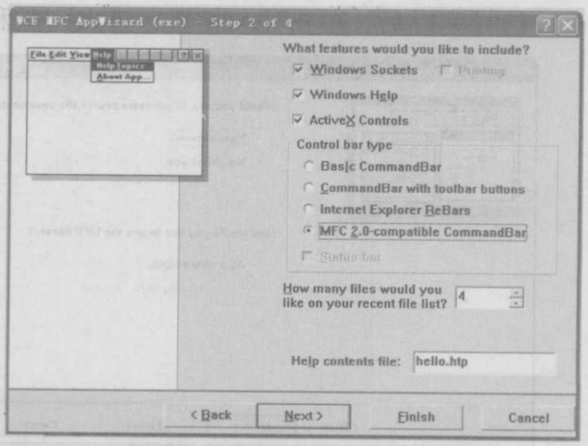


图4-41 用户特性选择界面

其中，如果打算使用TCP/IP网络通信，必须要选中Windows Sockets；选中Windows Help，可以在程序中使用联机帮助；ActiveX Controls则对使用ActiveX控件做出了相应的支持。

在这里有一点需要提及的是，在Windows CE环境下，应用程序不能拥有菜单，但是可以有一个变通的办法，使用命令条（Command Bar）。命令条类似于加强的工具条，在上面除了可以有工具按钮以外，还可以显示菜单和位图。这样就解决了问题。Control bar type一共有4个选项可以选择：

- 基本的命令条。
- 带有工具栏按钮的命令条。
- IE类型的工具条。
- 与MFC2.0兼容的命令条。

而下面的组合框决定了你打算在文件菜单中包含几个最近打开过的文件，如果不想包含，将数字调至零即可。在图4-41中可以看到，组合框中显示的数字为4，这就表示将会有4个最近被打开的文件被显示出来。

如果还想为程序提供联机帮助，那么在帮助目录文件后的文本框中输入帮助文件的名字即可。

进行完上述操作以后，单击Next按钮，进入到MFC类库的选择界面上。类库选择界面如图4-42所示。

如果打算让编译器为它所生成的源码加入注释，则选择Yes，这也会让编译器额外生成一个Readme.Txt，其内容是对该工程所有文件的一个简要说明。

下面的选项是询问使用MFC库的方法。其中As a shared DLL表示动态链接MFC库，目标Windows CE系统必须提供这些函数库，通常这是个减小可执行文件大小的好方法，但有时也会惹来麻烦。如果程序被使用不同语言的Windows CE系统的用户打开，那么文字编码上的差异可能会使程序运行异常，如果程序被放到没有包含MFC链接库的系统上执行，则应该把库文件作为应用程序报的一部分提供给用户。动态链接到MFC要求提供MFCnn.dll库文件，nn表示MFC的版本号。

如果选择静态链接库，在链接的时候，连接程序就会把相应的函数库加入到可执行文件中，Windows CE系统可以不提供这些动态链接库，但会使可执行文件尺寸变大，并且使内存利用率降低。

选择完以后，单击Next按钮，进入到类和文件名称设置界面，如图4-43所示。

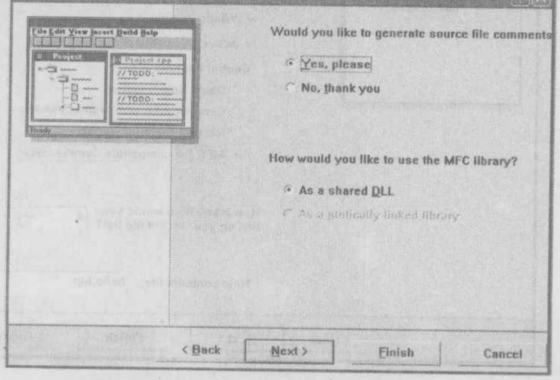


图4-42 MFC类库选择界面

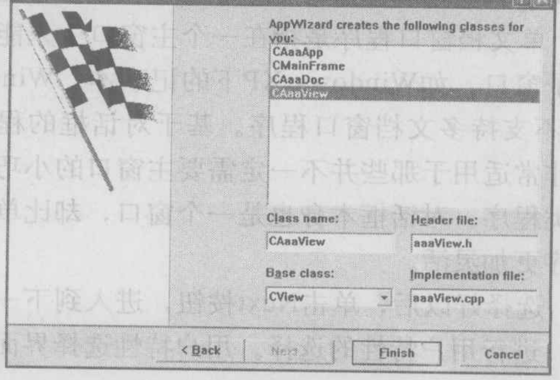


图4-43 类和文件名设置界面

在这一步中，可以自定义生成程序的文件名，包括原文件和头文件，还可以选择代码中的类是从哪个类继承的。选定CAaaView，单击基类里面的下拉列表，会出现几个选项：CEditView、CTreeView、CListView等，选择其中的任一个都会给程序带来不同的视图特征。同样，也可以定制代码中的其他类。

单击Finish按钮完成向导。一个简单的应用程序就生成了。

4.4.4 在构建好的MFC程序框架中添加消息

当建立起来一个应用程序时，只是建构了最初的框架，如果想让程序按照我们的意愿执行各种功能，就一定要处理来自Windows的各种消息。完成处理消息功能的基本单元是消息处理函数。在一个程序中，消息处理函数可能会有很多，如果一个一个地手工填写就太麻烦了，ClassWizard可以方便地完成这个任务。

在菜单中选择View→ClassWizard，此时会出现图4-44所示的界面。

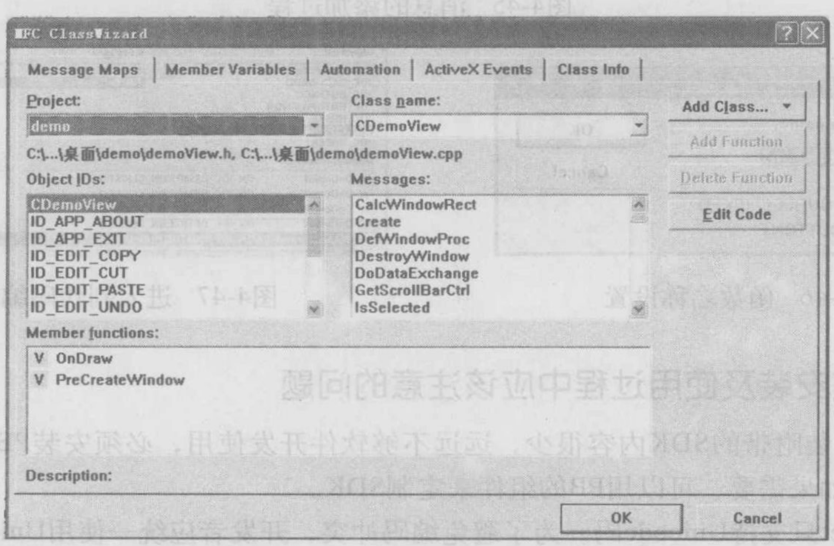


图4-44 ClassWizard界面

第一个选项卡用于制定程序的消息映射，中间左边是工程中定义过的标识符，右边是窗口收到的消息。例如，想处理用户按键事件，可以选择WM_KEYDOWN消息，单击Add Function按钮，消息处理函数就被加入到了当前的类中，一旦该消息被发送，此函数就会处理此消息。其中消息映射的绘制工作是由MFC体系结构自动完成的。

如果单击Object IDs，选择里面已有对象的标示，然后在Messages中选择BN_DOUBLECLICK ED，再单击右侧的Add Function按钮，表示把BN_DOUBLECLICKED消息和IDC_BUTTON关联起来，这样当双击该按钮的时候，就会激发BN_DOUBLECLICKED消息，以便进行相应动作的处理。此过程如图4-45所示。

当单击Add Function按钮以后，就会出现图4-46所示的画面。该窗口询问将要处理此对象事件的函数名，在设置好函数名称以后，单击OK按钮。此时会在Member functions中出现刚才新增加的函数。

选中该函数，然后单击右侧的Edit Code按钮就会进入到代码编辑界面，如图4-47所示。

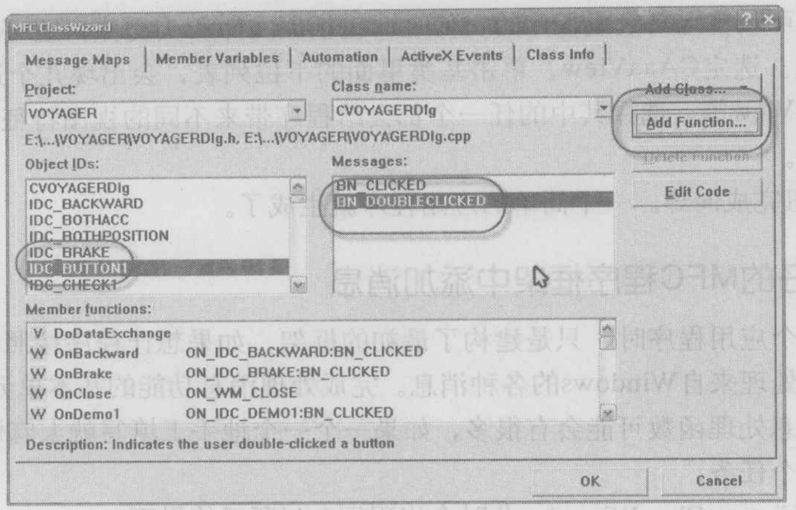


图4-45 消息的添加过程

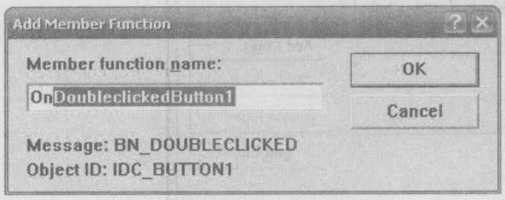


图4-46 函数名称设置

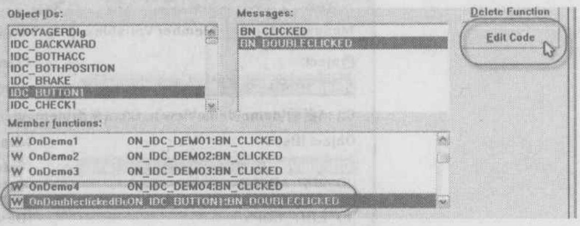


图4-47 进入到代码编辑界面

4.4.5 EVC在安装及使用过程中应该注意的问题

1）随EVC安装附带的SDK内容很少，远远不够软件开发使用，必须安装PB的全部SDK或者专门安装SDK。如果需要，可以用PB的组件来定制SDK。

2）WinCE.net只支持Unicode码。为了避免编码冲突，开发者应统一使用Unicode来实现。

3）EVC安装之后，并没有产生帮助文件的快捷方式，需要调出帮助文件时必须先打开EVC，再打开Help菜单。

4）安装、配置和编译EVC有时会出错。无论出于什么原因，如果第一次安装EVC失败了，那么以后再安装也很难成功，因为EVC的卸载也不是很干净，很多注册信息仍留在注册表中。解决办法是重装操作系统，当然，克隆系统镜像会更快。最好是一次安装成功。

5）EVC模拟器不支持中文，所以在EVC模拟器上运行的软件最好不要加入中文显示。

6）EVC的帮助文档写的不是很好，有些环境提供的函数，文档中却找不到说明，而有些帮助文档中支持的函数，在编译时EVC却不识别，这是微软比较大的一个BUG。只有具体靠编译运行才能检验是否正确，使用时候需要特别的注意。

4.5 ActiveSync

ActiveSync是微软公司推出的一款用于在PC和智能设备之间进行数据同步的一款软件。可以使用它进行如下一些操作：

1）同步PC的Outlook中所有的内容，如联系人、已经发送的邮件、收到的邮件等。

2）利用ActiveSync在PC上为Windows Mobile设备安装基于Windows Mobile的应用程序。

3）利用ActiveSync在PC上浏览Windows Mobile设备中的文件，并能够在PC和Windows Mobile设备之间进行文件的复制、移动等操作。

4）利用ActiveSync可以使得Windows Mobile设备共享PC的网络连接，进行网页信息的浏览。

所有Windows Mobile设备都内置ActiveSync工具，PC端的ActiveSync工具可以在微软网站免费下载，也可以在设备的随机光盘中找到。图4-48所示为ActiveSync软件成功连接Pocket PC时的画面；图4-49所示为通过ActiveSync软件浏览Pocket PC时的画面。

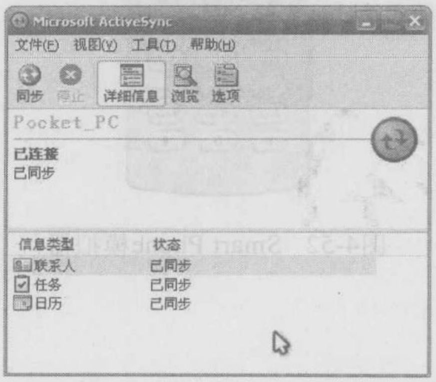


图4-48 ActiveSync软件成功连接时的画面

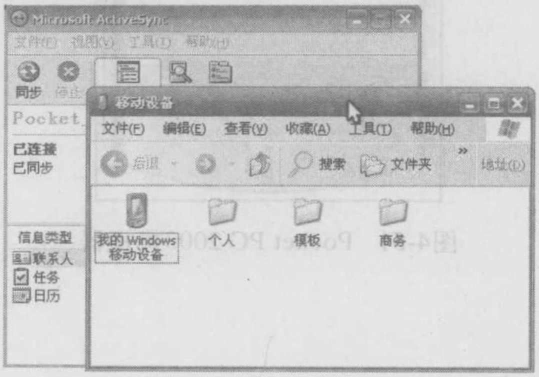


图4-49 通过ActiveSync浏览Pocket PC

4.6 模拟器

嵌入式程序与传统的PC程序有所不同，它们需要运行在嵌入式操作系统上，如Windows CE、Pocket PC、Smart Phone等。因此为了更好地仿真程序运行的环境，每一款嵌入式集成开发环境都集成了针对不同嵌入式设备的模拟器。

图4-50所示为Windows Embedded CE 6.0的模拟器；图4-51和图4-52所示为Pocket PC 2003和Smart Phone的模拟器。

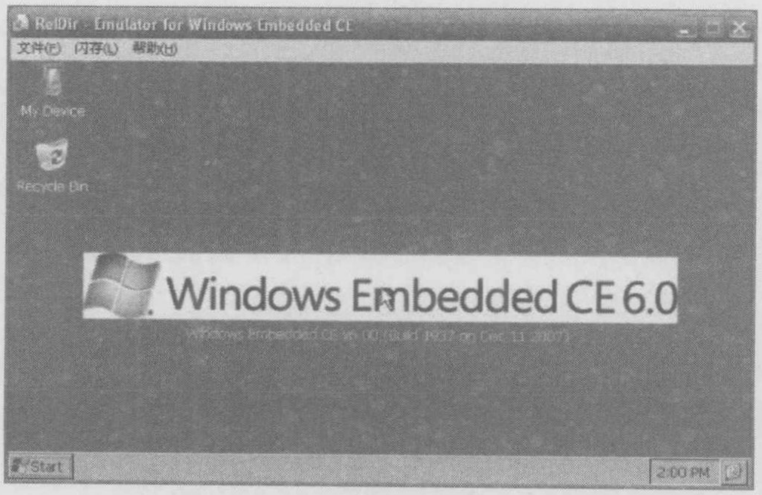


图4-50 Windows Embedded CE 6.0模拟器

图4-51 Pocket PC 2003模拟器

图4-52 Smart Phone模拟器

第5章　Windows Embedded CE 6.0中的内存管理

5.1　Windows Embedded CE 6.0中的RAM和ROM

在Windows Embedded CE 6.0程序设计中，内存管理是非常重要的一个环节。和传统PC比起来，Windows CE系统需要更加高效、节省地使用物理内存。尽管Windows CE主要被用在嵌入式系统上，但是在Windows XP和Windows Me中采用的内存管理API函数被几乎完整地保留下来，如在Windows CE中支持虚拟内存动态分配、支持局部和单独的堆空间和内存映射文件等。

和传统的PC不同，Windows CE设备一般没有磁盘驱动器，它的内存通常是由RAM和ROM组成的。一般情况下，ROM的内容在机器出厂的时候由厂商决定，它里面主要包括操作系统和自带的一些应用程序。例如，当买来一台Pocket PC以后，里面是会自带Pocket PC 2003操作系统，同时还会安装好Pocket Word、Pocket Excel、资源管理器、Media Player等一些应用软件，这些都存储在ROM中。在这里需要特殊说明的是，ROM中的程序可以被指定为本地运行，也就是说不需要装载到RAM中而直接运行。Windows CE中的RAM又可以分成可改变大小的程序内存和对象存储区域。

下面首先来看一下Windows Embedded CE 6.0中的RAM。

5.1.1　RAM

经常接触掌上电脑的朋友一定会注意到，在介绍设备性能的宣传资料中，经常会提到RAM和ROM。在Windows Embedded CE 6.0中，RAM主要是在内核和应用程序运行时使用。只要系统有电力供应，它便处于活动状态。因此，当Pocket PC长时间不用以后，存储在RAM里面的程序就会被清除。

Windows CE中的RAM分为程序区（也叫系统堆）和对象存储两个区域。其中的对象存储区域可以被看作一个永久的虚拟RAM盘。和传统的PC不同，Windows CE中的对象存储区在系统关闭以后仍旧可以保存文件。这是因为在移动设备，如Pocket PC中，当关闭电源时，系统并没有真正地停止运行，而是进入到了低功耗的挂起状态。之所以被设计成这样的运行模式，是因为根据嵌入式系统的自身特点，Windows CE系统被设计成了主电池和备份电池的双电池策略。当用户更换主电池的时候，系统的备用电池将自动地对对象存储区供电，使得对象存储区中的文件仍旧可以保存下来。当用户按下“重起”按钮以后，Windows CE内核会在RAM中寻找最近一次创建的对象存储，如果找到，就用这个对象存储重新启动。

RAM的另一个区域是程序区。这个区域就像PC的RAM一样。它存放所运行程序的堆和栈的空间。对象存储和程序区的界限是可移动的。在低内存的情况下，系统会请求用户把部分可用的存储对象空间提供给运行程序使用，以满足程序运行时对RAM的需求。

5.1.2 ROM

ROM的英文全称是Read Only Memory，中文名称是“只读存储器”。它是内置于计算机、Pocket PC等计算机设备上的永久存储设备，和RAM不同，这种存储设备即使在关闭电源以后，仍旧可以保存其中存储的信息。

在Windows CE中，存储在ROM中的程序可以被设计成本地执行的（XIP）。利用这种执行方式，程序可以直接在ROM中被执行。我们知道，一般的程序在运行的时候，首先需要被释放到RAM中，然后再开始执行。而采用了XIP执行方式，则具有以下优点：

- 程序执行的时候，不会占用在嵌入式系统中非常宝贵的RAM资源，因为它不需要被释放到RAM。
- 由于省去了释放到RAM的过程，因此程序启动的时间将大为缩短。
- 在小型嵌入式系统中如果采用XIP技术，则可以减少程序执行时对RAM的需求，这样就可以减少RAM的容量，会在一定程度上节约产品的成本。

但是，在具备上述优点的同时，XIP执行方式也带来了一定的问题。当程序在ROM中执行的时候，由于ROM的CPU访问速度远远低于RAM的访问速度，因此程序执行的速度相对较慢。正是基于上面的原因，当用Platform Builder创建一个对实时性要求很高的嵌入式系统内核时，一般不采用XIP执行方式。

需要注意的是，存储在闪存（Flash Memory）中和对象存储区（Object Store）的程序不能被本地执行，它们需要首先被复制到RAM中，然后再开始运行。

这里提到的XIP的英文全称是Execute in place，中文名称是“本地执行”。对于一般的存储设备而言，程序在执行之前需要被加载到主存中，如RAM或者闪存（Flash Memory），然后再被执行。而采用XIP执行方式，则可以省掉加载的过程。

如在传统的Linux中，需要启动程序来确定内核的位置和需要把内核的哪部分加载到RAM里面来运行，而在ucLinux（不支持虚拟内存的一种Linux系统）中，可以采用XIP技术让内核在存储器中直接被执行。这种方式非常适合小型嵌入式系统。因为可以把代码放在闪存中，在系统启动的时候直接运行它，这样就减少了RAM的容量，从而在一定程度上节约了成本。

5.1.3 RAM和ROM在实际设备中的体现

为了更好地理解RAM和ROM，这里以Pocket PC为例加以说明。当买来一个新的Pocket PC的时候，就会发现在里面已经内置好了Pocket Word、Pocket Excel、Media Player等应用软件。那么是否想过，这些软件和我们自己添加的游戏软件在安装位置上是否有什么不同呢？你是否遇到过Pocket PC断电以后（如长时间不用，电池电量耗光）自己安装的程序全部消失的现象呢？这些都和Pocket PC中的ROM和RAM有关。

一般地，Pocket PC在出厂的时候，就会把操作系统（如Pocket PC 2002或者Pocket PC 2003）和一些应用软件（如生产厂家设计的电源管理软件、媒体播放软件、蓝牙、红外等无线设备管理软件等）安装到Pocket PC的ROM里面，而自行安装的软件则通常会存储在RAM中（也可以手动地改变存储的位置，如内置的SD卡或者CF卡中）。根据前面介绍的ROM的特性，安装在ROM里面的软件断电以后自然仍旧可以被保留，而装在RAM里面的程序则会被自动地清除。因为RAM断电以后不具有保存文件的能力。图5-1为内置的Pocket Word截图，图5-2为内置的Pocket Excel截图。

为了防止由于意外断电导致的程序文件的丢失，微软公司在Pocket PC中安装了一个叫做Backup的程序。可以把已经安装好的程序完全备份到内置的SD卡或者CF卡上，这样当给Pocket PC充电以后，就可以通过这个软件把程序完全恢复过来，省掉了重新安装的麻烦。

在前面关于RAM的介绍中已经提到，在RAM里面分为两部分，一部分是程序存储区，一部分是对象存储区，两个区的大小是可以动态变化的。关于这一点，可以在Pocket PC中很直观地看到。图5-3为Pocket PC的内存显示画面；图5-4则显示了手工调整后的内存显示画面。

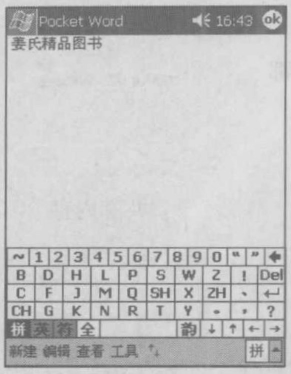


图5-1 内置的Pocket Word截图

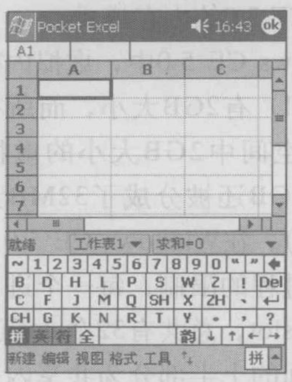


图5-2 内置的Pocket Excel截图

图5-3 Pocket PC中的内存控制

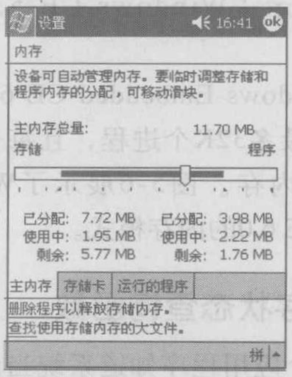


图5-4 调整后的Pocket PC中的内存

在图5-3、图5-4显示的图片中，内存控制条左边的“存储”对应在RAM中提到的对象存储区；右边的“程序”对应在RAM中提到的程序区。从上面这两幅图中，可以很直观地看出RAM中的对象存储区和程序区可以灵活地进行调整。

在台式电脑中，ROM中存放的BIOS，通常的大小是64~128KB。而在Windows CE系统中，ROM中存放的则是整个操作系统以及操作系统中绑定的引用程序，一般的大小为4~32MB。因此，可以把Windows CE下的ROM看作是一个很小的只读硬盘。

5.2 虚存

随着计算机技术的不断发展，应用程序的规模不断地扩大，对内存的需求也是越来越大。尽管在理论上可以依靠增加物理内存来缓解这一矛盾，但是这样就会增加计算机系统的成本。为了在成本和性能之间找到一个较好的平衡点，虚存技术应运而生。在了解虚存特点以前，首先看一

下Windows Embedded CE 6.0的内存模式。

5.2.1 Windows Embedded CE 6.0中的内存模式

和Windows CE 5.0相比，Windows Embedded CE 6.0在内存方面发生了非常大的变化。在了解这些变化以前，首先来看一下Windows CE 5.0的内存模式。

在Windows CE 5.0中，虚拟内存的内核空间（Kernel）有2GB大小，而所有进程则共享了用户空间中2GB大小的虚拟内存映射，同时这2GB还被分成了32MB大小的若干个槽（slot）。在Windows CE 5.0中，系统设置了32个进程的限制，每一个进程分配了一个32MB的slot，总共有32个slot可供分配。并且用户空间的上半部分为共享内存，并且这些共享内存能够被所有进程读写。

图5-5展示了Windows CE 5.0的内存模式。

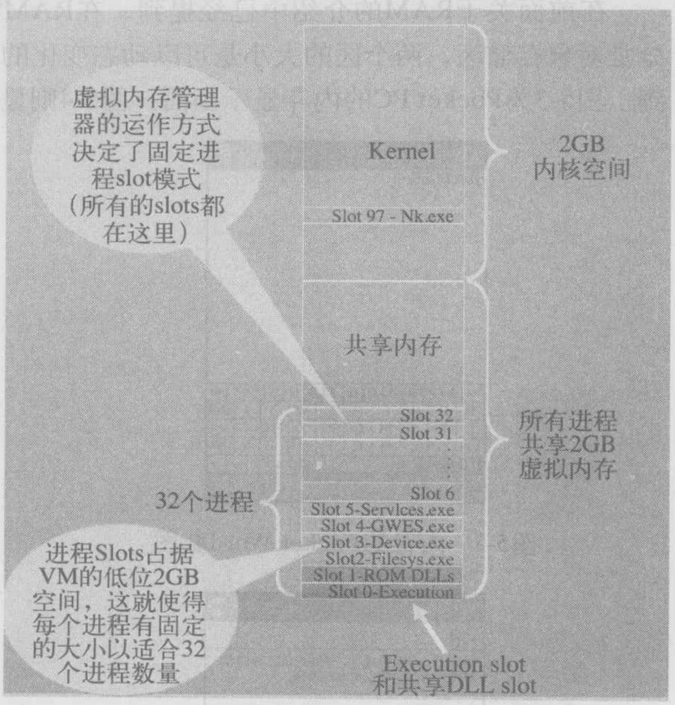


图5-5 Windows CE 5.0的内存模式

而在Windows Embedded CE 6.0中，系统则具备了最多32K个进程，且每个进程有2GB的虚拟内存。图5-6展示了Windows Embedded CE 6.0的内存模式。

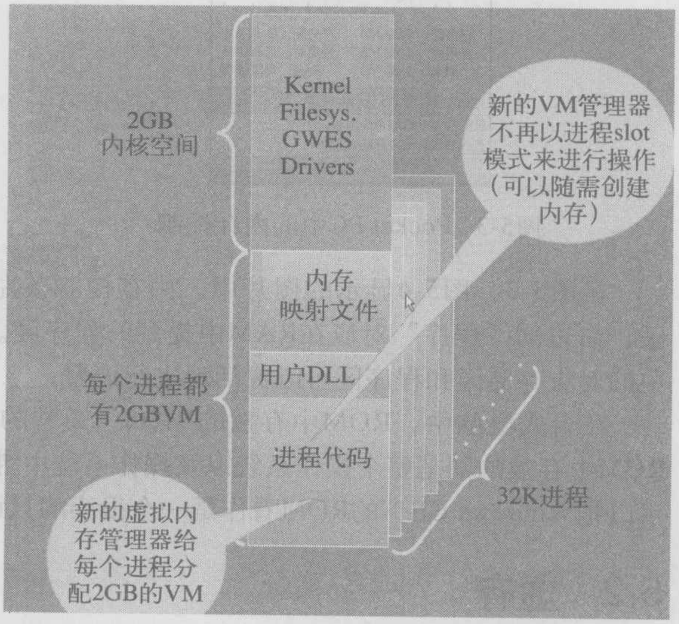


图5-6 Windows Embedded CE 6.0的内存模式

5.2.2 内存状态查询函数

如果一个应用程序知道系统当前的内存状态，就可以很好地管理可利用的资源。在Windows Embedded CE 6.0中可以使用GetSystemInfo和GlobalMemoryStatus函数来查询系统的内存状态。GetSystemInfo函数的函数原型为：

```
VOID GetSystemInfo (LPSYSTEM_
INFO lpSystemInfo)
```

该函数传递了一个指向SYSTEM_INFO结构的指针。SYSTEM_INFO的结构定义如下：

```
typedef struct {
    WORD wProcessorArchitecture;
  WORD wReserved;
```

```
    DWORD  dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD  dwActiveProcessorMask;
    DWORD  dwNumberOfProcessors;
    DWORD  dwProcessorType;
    DWORD  dwAllocationGranularity;
    WORD  wProcessorLevel;
    WORD  wProcessorRevision;
} SYSTEM_INFO
```

各个参数的含义为：

- wProcessorArchitecture：表示系统微处理器的类型。
- wReserved：表示处于reserved状态的页。
- dwPageSize：表示微处理器页面的大小。
- lpMinimumApplicationAddress：程序用到的最小虚存地址，在Windows Embedded CE 6.0中，该数值为0x10000。
- lpMaximumApplicationAddress：程序用到的最大虚存地址，在Windows Embedded CE 6.0中，该数值为0x7FFFFFFF。
- dwActiveProcessorMask：表示系统中装配了的中央处理器的掩码，如二进制0位是处理器0；31位是处理器31。
- dwNumberOfProcessors：表示系统中处理器的数目。
- dwProcessorType：表示处理器的类型。
- dwAllocationGranularity：表示一个完整的虚拟内存区域分配的界限。
- wProcessorLevel：表示CPU类型的特征。
- wProcessorRevision：表示模式（model）和芯片的步进级别（stepping level）。

第二个获得内存状态的函数GlobalMemoryStatus，其函数原型为：

void GlobalMemoryStatus(LPMEMORYSTATUS lpmst);

该函数返回了一个MEMORYSTATUS结构，该结构的定义为：

```
typedef struct {
    DWORD dwLength;
    DWORD dwMemoryLoad;
    DWORD dwTotalPhys;
    DWORD dwAvailPhys;
    DWORD dwTotalPageFile;
    DWORD dwAvailPageFile;
    DWORD dwTotalVirtual;
    DWORD dwAvailVirtual;
} MEMORYSTATUS;
```

各个参数的含义为：

- dwLength：表示MEMORYSTATUS结构的大小，在调用GlobalMemoryStatus函数前用sizeof()函数求得，即sizeof(MEMORYSTATUS)。

- dwMemoryLoad：返回一个介于0～100之间的值，用来指示当前系统内存的使用率。
- dwTotalPhys：表示RAM中有多少页被分配给了程序存储区RAM。
- dwAvailPhys：返回可被程序使用的虚拟内存页的数量，以字节为单位。
- dwTotalPageFile：显示可以存在页面文件中的字节数，因为Windows CE不支持页面文件，因此这个参数总为0。
- dwAvailPageFile：返回可用的页面文件大小，因为Windows CE不支持页面文件，因此这个参数总为0。
- dwTotalVirtual：返回调用进程的用户模式部分的全部可用虚拟地址空间，以字节（byte）为单位。
- dwAvailVirtual：返回调用进程的用户模式部分的实际自由可用的虚拟地址空间，以字节（byte）为单位。

在这里需要注意的是，dwTotalPhys和dwAvailPhys的数值不包括分配给对象存储空间（object store）的RAM值。

当在128MB RAM大小的Pocket PC上调用GlobalMemoryStatus函数时，会返回下面的信息：

```
dwMemoryLoad        0x18          (14)
dwTotalPhys         0x011ac000    (56,102,912)
dwAvailPhys         0x00B66000    (48,304,128)
dwTotalPageFile     0
dwAvailPageFile     0
dwTotalVirtual      0x02000000    (1,073,741, 824)
dwAvailVirtual      0x01e10000    (1,073,217, 536)
```

其中（56,102,912）表示有56,102,912大小的RAM被分配给了程序存储RAM，在这些程序存储RAM中，还剩余48,304,128可以被使用。在这里需要特别提及的是，通过这个函数的调用并不知道有72MB左右的空间被分配给了对象存储RAM，如果想得知对象存储RAM的大小，则需要使用GetStoreInformation函数。

dwTotalPageFile和dwAvailPageFile参数是0，表明页面文件不被Windows CE所支持。dwTotalVirtual参数则显示了Windows CE强制给程序的1GB的虚拟内存限制。其间，dwAvailVirtual参数则显示了只使用了1GB虚拟内存的一小部分，即1,073,741,824－1,073,217,536＝524,288。

5.2.3 Windows Embedded CE 6.0中的分页式虚拟存储

同Windows CE 5.0一样，在Windows Embedded CE 6.0中，系统仍旧采取的是分页式虚拟存储（Paged Virtual Memory System），在这种分页式虚拟存储中，微处理器管理的最小内存单元叫做页。当一个应用程序需要访问一个页的时候，微处理器就会根据需要，把虚拟地址转换成RAM或者ROM中的实际物理地址。在Windows Embedded CE 6.0中，一个页的大小为4096字节，这一点和32位的桌面操作系统Windows XP和Windows Vista是一样的。而在早期版本的Windows CE中（如Windows CE 2.12），一个页的大小则为1024个字节。

在Windows Embedded CE 6.0中，虚拟页面（Virtual Page）有三种状态，分别为未使用的（free）、保留的（reserved）和占用的（committed）。它们的含义分别为：

- 未使用的虚拟页面：未使用的虚拟页面就是指可以被分配的。

- 保留的虚拟页面：保留的虚拟页面就是指该页面已经被预订，它的虚拟地址空间不能再被分配给操作系统或者另外一个进程中的线程。因为保留的页面没有与实际的物理内存对应起来，因此仍旧不可以被程序使用。
- 占用的虚拟页面：占用的虚拟页面是指已经被一个程序占用、保留的虚拟页面。它已经被直接映射到了实际的物理地址空间。

5.2.4 与虚拟存储相关的函数

在Windows CE中，分配和预订保留虚拟内存的函数是：

```
LPVOID VirtualAlloc(LPVOID lpAddress, DWORD dwSize, DWORD flAllocationType,
                    DWORD flProtect);
```

VirtualAlloc函数的功能类似于C++语言中的new操作符，这个函数是底层的分配虚拟地址空间的函数。各个参数的含义为：

- lpAddress：参数lpAddress为虚拟地址空间的首地址。如果这个参数设置成0，那么内核将自动查找一个符合要求的首地址。
- dwSize：参数dwSize为空间的大小。
- flAllocationType：参数flAllocationType为分配类型，它可以是下面这些类型的一个组合：
  - MEM_COMMIT：为程序分配将要使用的内存空间。
  - MEM_AUTO_COMMIT：内存块中的页被自动地提交。
  - MEM_RESERVE：为程序分配将要被提交的虚拟地址空间（也就是被保留的虚拟地址）。
  - MEM_TOP_DOWN：告诉系统在为程序分配虚拟地址空间的时候，尽量从空闲区域的顶部开始分配。
- flProtect：参数flProtect为分配的虚拟地址空间设置保护标志。它有以下几种模式：
  - PAGE_READONLY：区域被设置成只读的。当一个应用程序试图向这个区域中的页写入数据的时候，这个访问控制策略将发挥作用。
  - PAGE_READWRITE：区域被设置成可读、可写的。
  - PAGE_EXECUTE：区域中包含可以被系统执行的代码。当试图从这个区域读取或者写入数据的时候，会引起访问冲突。
  - PAGE_EXECUTE_READ：区域中包含可以执行的代码，并且只允许应用程序从这个区域读取数据。
  - PAGE_EXECUTE_READWRITE：区域中包含可以执行的代码，并且程序可以读取和写入数据。
  - PAGE_GUARD：这个属性需要和其他的属性配合使用。当首次访问虚拟空间的时候，将会引起STATUS_GUARD_PAGE异常。当再次访问这个区域的时候，就会按照其他的属性来执行。这也是为什么需要和其他属性配合使用的原因。
  - PAGE_NOACCESS：不允许任何程序访问虚拟内存空间。
  - PAGE_NOCACHE：停用页面的高速缓存。

PAGE_GUARD和PAGE_NOCACHE可以和其他属性组合使用，以便更好地、更灵活地定义页面的保护标志（访问策略）。其中PAGE_NOCACHE属性主要是供那些驱动程序开发人员来使

用的。

Windows CE中释放虚拟内存的函数是：

```
BOOL VirtualFree(LPVOID lpAddress, DWORD dwSize, DWORD dwFreeType);
```

各个参数的含义为：

- lpAddress：参数lpAddress为需要释放的虚拟内存的首地址。
- dwSize：参数dwSize为需要释放的虚拟内存的大小。
- dwFreeType：参数dwFreeType为指定的释放类型，它主要有下面的两种：
  - MEM_DECOMMIT：当参数为MEM_DECOMMIT时，函数将释放和这个虚拟内存空间相映射的物理内存，但是还保留着这块虚拟地址空间。
  - MEM_RELEASE：当参数为MEM_RELEASE时，函数在释放物理内存的同时，也释放掉这块虚拟地址空间。

在这里，有一个地方需要引起注意，那就是当使用VirtualFree函数释放虚拟内存的时候，要保证内存中的所有页面都处于同一个状态。也就是说，要么所有的页面都是提交、占用的状态（committed），要么都是预定保留状态（reserved）。

当出现部分页面为保留状态，部分页面为提交状态的时候，VirtualFree函数就会调用失败。当虚拟内存中的页面有的已经被提交，有的没有被提交的时候，必须先调用此函数并传递撤销提交标志，先将提交的这部分物理内存释放掉，然后再调用此函数调用释放标志，释放整个的虚拟地址空间。

当想更改虚拟内存空间的访问权限的时候，可以调用下面的这个函数：

```
BOOL VirtualProtect (LPVOID lpAddress, DWORD dwSize, DWORD flNewProtect, PDWORD
                    lpflOldProtect);
```

各个参数的含义为：

- lpAddress：参数lpAddress为需要重新定义访问权限的虚拟内存的首地址。
- dwSize：参数dwSize为虚拟内存空间的大小。
- flNewProtect：参数flNewProtect为新定义的访问权限。
- lpflOldProtect：和参数flNewProtect相对应，lpflOldProtect为原来虚拟地址空间中第一个页面的保护标志。

当想查询一个虚拟地址空间的保护权限的时候，可以使用下面的函数：

```
DWORD VirtualQuery (LPCVOID lpAddress, PMEMORY_BASIC_INFORMATION lpBuffer,
                   DWORD dwLength);
```

各个参数的含义为：

- lpAddress：参数lpAddress为被查询虚拟地址空间的首地址。
- lpBuffer：参数lpBuffer为指向PMEMORY_BASIC_INFORMATION结构的指针。
- dwLength：参数dwLength为PMEMORY_BASIC_INFORMATION结构的大小。

那么，究竟什么是PMEMORY_BASIC_INFORMATION结构呢？下面就来看一下。PMEMORY_BASIC_INFORMATION的结构为：

```
typedef struct_MEMORY_BAISC_INFORMATION{
    PVOID BaseAddress;
```

```
    PVOID AllocationBase;
    PVOID AllocationProtect;
    PVOID RegionSize;
    PVOID State;
    PVOID Protect;
    PVOID Type;
}PMEMORY_BASIC_INFORMATION;
```

各个参数的含义为：

- BaseAddress：参数BaseAddress是查询函数VirtualQuery的基地址。
- AllocationBase：参数AllocationBase是用VirtualAlloc函数分配内存时，实际分配的基地址。需要特别说明的是，BaseAddress一定包含在AllocationBase分配的范围内。
- AllocationProtect：参数AllocationProtect是指分配该页面时页面的一些属性，如PAGE_READWRITE、PAGE_EXECUTE等。
- RegionSize：参数RegionSize是指从BaseAddress开始，具有相同属性页面的大小。
- State：参数State指的是页面的状态，有释放、预定和提交三种方式。
- Protect：参数Protect是指当前虚拟地址空间的保护标志。
- Type：参数Type是指内存空间的类型。主要的类型有以下几种：
  - MEM_PRIVATE：MEM_PRIVATE表示应用程序空间中包含私有数据。
  - MEM_MAPPED：MEM_MAPPED表示虚拟地址空间和内存映射文件向匹配。
  - MEM_IMAGE：MEM_IMAGE表示虚拟地址空间和一个EXE或者DLL类型的文件相匹配。

为了更好地理解VirualQuery函数，再来看看当一个程序通过VirtualQuery函数预定16,384字节大小的内存空间时的情况。首先，系统将会从0xA0000地址开始预定相应的地址空间，然后从2048字节的地方开始提交9216个字节大小的9个页面。这一过程可以参考图5-7。

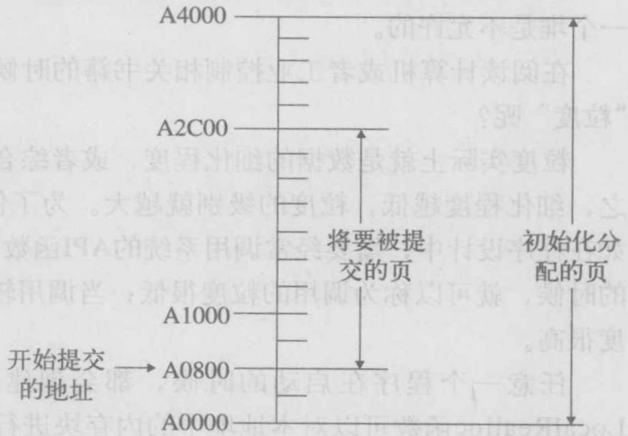


图5-7 VirualQuery函数的图示说明

当VirualQuery函数中的lpAddress指针指向0xA1000地址的时候，各个参数的返回值分别为：

```
BaseAddress              0xA1000
AllocationProtect        PAGE_NOACCESS
AllocationBase           0xA0000
RegionSize               0x1C00
State                    MEM_COMMIT
Protect                  PAGE_READWRITE
Type                     MEM_PRIVATE
```

5.3 堆

从本质上说，堆就是一段连续的、相对较大的虚拟地址空间。那么，堆和前面提到的虚存又有什么区别和联系呢？下面就来看一下。

5.3.1 Windows Embedded CE 6.0中堆的特性

前面介绍了分页式虚拟存储，由于这种方式是按照“页”的大小对内存进行分配的（粒度相对较大），因此对于大多数应用程序来说，其执行的效率相对较低。一般地，为了最大地提高内存的使用效率，一个应用程序应该可以任意分配或者释放一个字节的内存空间，或者至少8个字节的内存空间。为了实现这种高效的方式，Windows CE系统中使用了“堆”。使用“堆”，可以以字节为单位来申请和释放内存，很显然，这种方式的粒度比分页式虚拟存储要小得多。

在Windows CE系统中，根据程序对内存的需求，系统将提供相应的函数，以便在堆中分配和释放小的内存块。同时，系统还将会随着所运行程序的需求自动更改堆的大小。与经常使用的Windows XP不同，Windows CE只允许在堆中程序申请静态的（不能够移动的）内存块。尽管这一特性简化了对堆中内存块的控制，但是会随着对堆中内存块的不断分配和释放，产生很多碎片。这样，当需要分配一个大一点的连续内存块时，本来堆内的空闲空间加起来足够用，但是这些空闲的内存空间彼此之间是分割的，因此仍旧不符合要求。这就会出现只有当整个堆被释放以后，才可以分配到合适的内存的现象。

在Windows CE中，每个堆都是由一个信号量来控制访问顺序的，两个进程如果想同时访问一个堆是不允许的。

在阅读计算机或者工业控制相关书籍的时候，经常会遇到“粒度”这个词，那么究竟什么是“粒度”呢？

粒度实际上就是数据的细化程度，或者综合程度的级别。细化的程序越高，粒度就越小；反之，细化程度越低，粒度的级别就越大。为了便于理解，再来看一下粒度在具体环境中的含义。如在程序设计中，需要经常调用系统的API函数，当对API函数的调用较少，或者调用的频率较低的时候，就可以称为调用的粒度很低；当调用较多，或者调用的频率较高的时候，就可以称为粒度很高。

任意一个程序在启动的时候，都会创建一个本地堆。通过LocalAlloc、LocalFree和LocalRealloc函数可以对本地堆中的内存块进行分配、释放和调整大小。同时，一个程序也可以创建任意数量的独立堆。下面就来分别看一下本地堆和独立堆。

5.3.2 本地堆

在不同版本的Windows CE中，系统可以创建不同大小的“本地堆”（Local Heap）。对于程序开发人员来说，本地堆的初始化大小并不是非常重要，因为后续可以根据程序执行时的需要来动态地增加堆的大小。不过在这里要特别提及的是，由于为这些本地堆分配的地址空间在虚拟地址上并不一定会相邻，因此难免会产生一些“碎片”。在Windows Embedded CE 6.0中，分配内存、释放内存和重定义内存大小的函数只是Win32中，与本地堆相关函数的一个子集。

可以通过下面的这个函数在本地堆中分配内存块：

```
HLOCAL LocalAlloc(UINT uFlags, UINT uBytes);
```

这个函数将返回一个指向本地内存块的句柄；参数uFlags描述的是分配内存块的类型；参数uBytes则表示所分配内存块的大小。uFlags的三种类型为：

- LMEM_FIXED：表示在本地堆中分配固定的内存块。
- LMEM_ZEROINT：把分配的内存块中的数据全部设置为0。
- LPTR：包含了LMEM_FIXED和LMEM_ZEROINT两种类型所具有的特征。

如果需要释放本地堆中的内存块时，则需要下面这个函数：

```
HLOCAL LocalFree(HLOCAL hMem);
```

当分配内存块成功的时候，这个函数将返回一个NULL值。

重新设置内存块大小的函数为：

```
HLOCAL LocalReAlloc(HLOCAL hMem, UNIT uBytes, UNIT uFlag);
```

各个参数的含义为：

- hMem：由LocalAlloc函数返回的一个指针。
- uBytes：被重新设置的内存块的大小。
- uFlags：被重新分配的内存块的特性。

查询内存块的大小则可以用下面的函数：

```
UNIT LocalSize (HLOCAL hMem);
```

需要特别提到的是，在Windows中，句柄是和对象一一对应的32位无符号整数值。对象可以映射到唯一的句柄，句柄也可以映射到唯一的对象。

在Windows系统中，Windows需要向程序设计人员提供相应的编程接口，以便程序设计人员访问、创建或者销毁对象。但是出于系统封装的考虑，Windows又不想向程序设计人员提供指针，因为指针包含了大量的信息。首先，指针给出了对象存储的确切位置；其次，程序设计人员要想操作指针，就必须知道指针内部的结构特征，这就是说，操作系统必须向程序员暴露相应的内部数据结构，这些都是系统的不稳定因素，都极大地破坏了系统的封装性。

我们比较熟悉的COM技术向用户隐藏了数据，只暴露了接口，并且只允许程序设计人员按照接口定义的方法操作数据。而句柄这种方式则允许按照自己的方式直接操作数据。既满足了Windows系统的封装性，实现了对数据的隐藏，又满足了程序设计人员在程序设计中的需要。

5.3.3 独立堆

为了避免本地堆中出现的碎片现象，当在一段时间里需要一段连续的空间时，更好的方式是创建一个独立堆，如经常使用的文本编辑器就是为每一个文件创建一个独立堆。当文件被打开或者被关闭的时候，这些独立堆将被创建或者释放。

在这里有一点需要注意的是，Windows CE中的堆和其余Windows系统中的堆具有相同的API函数，但是也有一点小小的不同，那就是Windows CE中不支持HEAP_GENERATE_EXCEPTIONS标志。

1. 创建堆

可以通过下面的函数来创建堆：

```
HANDLE HeapCreate (DWORD flOptions, DWORD dwInitialSize, DWORD dwMaximumSize);
```

在Windows CE系统中，第一个参数flOptions可以为NULL，或者为HEAP_NO_SERIALIZE；第二个参数dwInitialSize指定的是开始创建堆时提交的物理内存的大小，dwMaximumSize则表示创建的堆的空间的最大值。在这里，有以下几点需要加以注意：

（1）dwMaximumSize决定了在虚拟地址空间中为堆保留了多少页。如果在程序设计中想让Windows决定保留多少页，就可以把这个值设置成0。

（2）堆的默认大小是188KB。

（3）dwInitialSize决定了有多少保留页可以被立即提交。当把这个值设置成0的时候，堆将被初始化设置成提交一个页。

当这个函数调用成功的时候，函数将返回堆的句柄。在堆中分配内存块需要遍历整个堆的内存块链表，其目的就是寻找一个可以符合要求的内存块。在找到符合要求的内存块以后，就将这个内存块标记为已经分配，然后将这个内存块的信息添加到内存块链表中。

Windows CE中的HeapCreate函数只是创建了一个堆，而没有给这个堆分配或者保留任何的内存空间，只有在分配堆的第一个块的时候，Windows CE系统才会保留或者提交所需的内存空间。

2. 在独立堆中分配内存

可以通过下面的函数实现在独立堆中对内存的分配：

```
LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes);
```

需要注意的是，这个函数的返回类型是指针，而不是句柄。这一点和LocalAlloc函数不太一样。同Windows XP和Windows Me一样，Windows CE 中的独立堆分配的也是固定内存块。下面来看看各个参数的含义：

- hHeap：参数hHeap为指定堆的句柄。
- dwFlags：参数dwFlags为标志，可选值为HEAP_NO_SERIALIZE和HEAP_ZERO_MEMORY。
- dwBytes：参数dwBytes为指定分配的大小，它是以字节为单位的。

3. 在独立堆中释放内存

在独立堆中释放内存需要下面的函数：

```
BOOL HeapFree (HANDLE hHeap,DWORD dwFlags, LPVOID lpMem);
```

各个参数的含义为：

- hHeap：参数hHeap为指定堆的句柄。
- dwFlags：参数dwFlags为标志，其值只允许为HEAP_NO_SERIALIZE。
- lpMem：参数lpMem为指向被释放内存块的指针。

4. 在独立堆中重新分配内存的大小

在独立堆中重新分配内存大小的函数为：

```
LPVOID HeapReAlloc (HANDLE hHeap, DWORD dwFlags, LPVOID lpMem, DWORD dwBytes);
```

各个参数的含义为：

- hHeap：参数hHeap为指定堆的句柄。
- dwFlags：参数dwFlags可以是下面三个参数中的任意一个。
  - HEAP_NO_SERIALIZE。

- HEAP_REALLOC_IN_PLACE_ONLY：当没有找到指定的内存块时，这个参数将会通知系统。
- HEAP_ZERO_MEMORY。

- lpMem：参数lpMem为指向重定义大小的内存块的指针。
- dwBytes：参数dwBytes为重新分配的内存块的大小。

当这个函数被成功执行以后，将返回一个指向内存块的指针；否则，将返回NULL值。

5. 在独立堆中查询堆的大小

可以用下面这个函数来查询堆的大小：

```
DWORD HeapSize (HANDLE hHeap, DWORD dwFlags, LPCVOID lpMem);
```

各个参数的含义为：

- hHeap：参数hHeap为指定堆的句柄。
- dwFlags：参数dwFlags为标志，只有HEAP_NO_SERIALIZE一个选项。
- lpMem：参数lpMem为指向内存块的指针。

6. 释放独立堆

如果想要释放一个独立堆，则可以通过下面的函数：

```
BOOL HeapDestroy (HANDLE hHeap)
```

参数的hHeap为指定堆的句柄。当这个函数调用成功的时候，函数返回True，否则，返回False。在释放堆以前，不需要把堆中的内存块释放掉。

5.4 栈

从前面对堆的介绍中可以知道，堆是一段连续的虚拟地址空间。同样，栈（Stack）也是一段连续的虚拟地址空间。但是和堆相比起来，栈的空间要相对小一些，同时栈是专为函数使用而设计的，是用来存放函数变量的。同时，系统也使用栈来存放函数的返回地址及异常处理中处理器所处的状态。

在Windows CE系统中，系统将为每一个线程创建一个独立的栈，创建的栈的默认最大值为56KB。这就是说根据需要，每一个进程中的独立线程都可以把它的栈空间最大增长到56KB。和堆不同，栈空间在增加的时候，是从高地址向低地址的。

当栈空间逐渐变小时，Windows CE系统将进入到低内存状态（low-memory condition）。栈空间的大小可以在程序链接的时候指定，最大可以达到1MB。在这里需要注意的是，默认栈空间的大小也是其他独立线程栈空间的大小。当一个应用程序将要调用一个函数时，Windows CE系统将立即提交满足要求的页。但是当发现没有足够可用的物理RAM时，这个线程就会被挂起。如果在一个给定的时间内，这个内存需求得不到响应，就是出现了系统异常。

因此，在程序设计的时候需要注意，在实际开发中尽量不要在栈中分配很多、很大的内存块。因为如果分配的内存块超过了一定的大小，就会引起非法的访问，同时内核将会立即中止该进程。根据经验，最好是在堆中分配大的内存块，并且一定要在使用以后把堆释放掉，或者也可以通过在创建线程时指定栈的大小来避免这个问题。

5.5 静态数据块

编写的C或者C++程序都会在程序载入时预定义一些内存块，以便为它们自动地分配空间。这些预定义的内存块中包含了静态申请的字符串、缓冲区、全局变量以及和这个应用程序静态连接的库函数。同样，Windows CE中的静态数据块也是Windows CE中一个应用程序启动时候装载的内存块。这些内存块中也包含了字符串、缓冲区和程序需要的静态数据。在Windows CE中，这种方式可以最大限度地利用每一个字节的RAM。

在Windows CE中，系统会为每一个应用程序申请两个RAM块，分别为：

- 用于存放可读/可写数据。
- 用于存放只读数据。

由于这些区域都是以页为单位来分配内存的，所以为了节省存储空间，应该尽量使静态数据段接近但不要超出页的范围。如果在静态数据段中还有一些空间，就可以把需要动态申请的1个或2个缓冲区移到静态数据区里面，而不是直接动态地分配这些缓冲区空间。

在编写一个基于ROM的程序的时候，最好能尽可能多地把数据放在只读静态数据区中。在Windows CE中，系统不会为一个基于ROM的应用程序分配RAM中的空间。而是会把ROM中的页直接和虚存地址相映射。这种方式实际上就为开发人员提供了一个可自由运用的只读空间，而不用考虑应用程序对RAM的需求所带来的影响。

5.6 Windows CE下的Bootloader

在这个实验中，将了解到Windows CE的启动过程，进一步了解Windows CE的一些特性。首先来看一下这个实验的实验目的和实验内容。

5.6.1 Bootloader简介

简单地说，Bootloader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，可以完成初始化硬件设备、建立内存空间的映射图、与内核镜像建立通信通道和调试通道等工作，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。通常来讲，Bootloader是严重依赖于硬件而实现的，特别是在嵌入式系统中（有些部分还不可避免地使用汇编语言实现）。因此，在嵌入式的世界里几乎不可能建立一个通用的Bootloader。所以针对特定的处理器，必须专门来定制Bootloader，以实现不同的功能需求。

图5-8展示了Bootloader和其他一些结构的关系。Bootloader通常存储在目标平台的非易失存储介质中，如ROM或者Flash中。

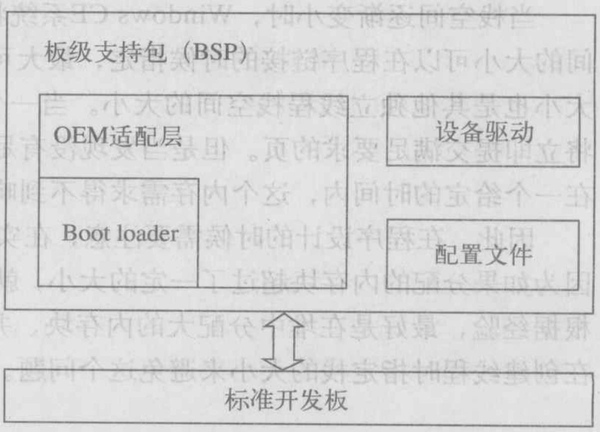


图5-8 Bootloader和其他一些结构的关系

在这里提到的Flash，实际上是指Flash ROM。众所周知，ROM是只读存储器。在早期的计算机发展中，BIOS信息都是被写在ROM中的。但是这就带来了一个问题，ROM中包含的资料是在ROM

的制造工序中，在工厂里用特殊的方法烧录进去的，由于ROM的特性导致其中的内容只能读不能改。因此烧录进去以后，用户只能验证写入的资料是否正确，而不能再作任何修改。当发现资料有错误的时候，则只有舍弃不用，重新订做一份。

随着技术的不断进步，人们又找到了一种非常好的替代方案，这就是采用Flash ROM。这种芯片的特点是存储在其中的信息可以通过一个专用的刷新程序进行重写。同时，在掉电的情况下存储在里面的资料也不丢失。正是由于这些特性，使得Flash ROM被应用在越来越多的产品中。目前的主板BOIS信息都是存储在Flash ROM中的。

5.6.2 Bootloader的主要功能

Bootloader的主要功能为初始化硬件设备、建立内存空间映射图，以便将系统的软硬件环境带到一个合适的状态下。Bootloader运行在操作系统之前，它的最终的目的是为调用操作系统内核准备好环境。

Bootloader分为两种操作模式，一种是启动模式（Bootloading），一种是下载模式（Downloading）。下面分别看一下。

启动加载（Bootloading）模式：即Bootloader自动加载系统的过程，整个过程并没有开发者和用户的介入。在正常条件下，必须保证这种启动方式的有效。在嵌入式产品发布的时候，Bootloader必须直接工作在这种模式下以保证系统的正常运行。

下载（Downloading）模式：在这种模式下，目标机上的Bootloader 将通过串口连接或网络连接等通信手段从主机（Host）下载文件。例如，下载内核映像和根文件系统映像等。从主机下载的文件通常首先被Bootloader保存到目标机的RAM中，然后再被Bootloader写到目标机上的Flash类固态存储设备中。有的系统还将启动各种通信方法。Bootloader的这种模式通常在第一次安装内核与根文件系统时被使用。此外，以后的系统更新也会使用Bootloader的这种工作模式。工作于这种模式下的Bootloader通常都会向它的终端用户提供一个简单的命令行接口，并可以执行简单的交互。

5.6.3 Bootloader的结构

看完了Bootloader的操作模式，再来看看Bootloader的结构框架。Bootloader分为Stage1和Stage2两部分。其中Stage1也被称为NBoot，Stage2也被称为EBoot。

Stage1（NBoot）的主要作用为：

- 硬件设备的初始化。
- 为加载Stage2准备RAM空间。
- 复制Stage2到RAM中。
- 设置好堆栈。
- 跳转到Stage2的入口。

Stage2（EBoot）的主要作用为：

- 初始化本阶段要用到的硬件设备。
- 检测系统的内存映射（memory map）。
- 将OS内核和根文件系统从Flash读取到RAM中。

• 为内核设置启动参数。

• 启动内核（调用内核）。

在Windows CE中，NBoot存储在Nand Flash的Block0中，TOC存储在Nand Flash的Block1中，EBoot存储在Nand Flash的Block2中。这里面提到的 TOC是整个Nand Flash中存储内容的一个列表，这里面储存了有关启动和系统内核的一些相关存储信息，它由NBoot里的函数读取。之后NBoot会按照读取的内容进行配置和跳转到不同的地址。使用文本查看工具可以看到TOC的主要组织形式。图5-9直观地展示了各自不同的存储位置。

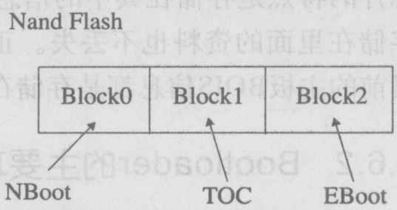


图5-9 不同的存储位置

在前面提到了Nand Flash这个名词，那么就来简单地看一下什么是Nand Flash。在这里提到的Nand是市场上两种非易失闪存技术之一（另外一种是NOR），Nand Flash自然就是采用了Nand技术的闪存芯片。Nand Flash结构是由东芝公司发明的，它强调降低每比特的成本，拥有更高的性能，并且要像磁盘一样可以通过接口轻松升级。

和NOR Flash适合存储少量的数据不同，Nand Flash更适合存储高数据存储秘密的数据。这是因为Nand结构能够提供更高的单元密度。同时，Nand Flash的写入速度和擦除的速度也很快。而应用Nand的困难则在于需要特殊的接口来管理Nand Flash。

第6章 Windows Embedded CE 6.0中的进程和线程

6.1 Windows Embedded CE 6.0中的进程

同经常使用的Windows XP一样，Windows Embedded CE 6.0是一种多任务、多线程的操作系统。我们知道，进程和线程是一个内核最主要的组成部分之一。任何一个软件都离不开内核的有效支持，只有掌握了进程和线程的基本知识，才能够很好地利用内核的特性，编写出高效率的嵌入式软件。在这一章中，将介绍进程、线程和指令模块等相关知识，以便对Windows Embedded CE 6.0有一个更深入的了解。

当打开两个Excel文档的时候，系统就会运行两个独立的进程。一个进程的运行目标就是执行它所对应的程序。如果没有程序，进程的存在就没有任何意义。

那么，进程和应用程序之间有什么样的关系，到底什么是进程呢？下面就来看一下。

6.1.1 进程概述

进程（Process）是一个具有一定独立功能的程序在一个数据集合上一次动态执行的过程，是一个正在运行的应用程序的实例。在早期版本的Windows CE中，系统最大允许32个独立的进程同时运行。而到了Windows Embedded CE 6.0中，系统则取消了其在数量上的限制。

进程具有动态性、并发性、独立性和结构化的特点。各自的含义为：

- 动态性：动态性指的是进程具有动态的空间地址，同时空间的大小和内容都是动态变化的。
- 并发性：并发性也被称为异步性，是指从宏观上看各个进程是同时运行的。
- 独立性：独立性是指各个进程的地址空间相互独立，只有采用进程间通信的方式才能相互影响。关于进程间通信的知识，将在本章的后面介绍。
- 结构化：结构化是指进程地址空间的结构化的划分，如被划分为代码段、数据段等。

在本章的前面曾经提到，进程和程序的关系非常密切，那么两者之间到底有什么样的联系和区别呢？

正如前面所述，进程是应用程序的一个实例，其运行的目标就是执行它所对应的应用程序，失去了程序这个载体，再谈进程就毫无意义。

进程和程序的区别主要体现在：

- 程序可以被放在存储设备，如Pocket PC中的RAM或者SD卡中进行长期的保存，而进程是有生命周期的。
- 一个程序可以对应多个进程，但是一个进程只能对应一个程序。例如，当打开两份不同的Power Point进行幻灯片演示的时候，就会对应两个不同的进程。
- 程序是静态的，进程是动态的。

同Windows XP中的进程不同，Windows CE中的进程一般包含较少的状态信息。因为

Windows CE不支持当前目录的概念，因此每一个单独的进程不需要保存这些状态信息；Windows CE不能保存一系列的环境变量，因此进程中不需要保存与环境变量有关的信息；Windows CE不支持句柄继承，所以也不需要保存这些相关的信息。由于以上提到的种种原因，Windows CE中的进程结构和Windows XP相比要简单得多。

同时在Windows CE中，许多进程或者与进程相关的函数都不被系统所支持。这是因为Windows CE系统不支持那些被Windows XP所支持的特性。因为Windows CE不支持环境变量，因此Win32函数中和环境变量相关的部分在Windows CE中都不存在；还有很多和Windows XP不同的地方，在这里就不一一介绍了。如果在程序设计中需要使用相关的函数，可参考MSDN等技术文档。

在这里有一点需要特别指出，一个进程中至少包含一个线程，且这些线程在宏观上都是同时运行的。但是从微观角度来看，实际上在Windows CE系统中，同一时刻只有一个线程来运行，只不过各个线程之间的切换时间非常的短，因此从整体上看就好像有多个线程在同时运行。

同时，在Windows Embedded CE 6.0中还有一些进程属于系统进程，这些进程与Windows CE 5.0相比除了名称上有一定变化以外，在功能上并没有什么大的不同。表6-1展示了Windows Embedded CE 6.0中主要的系统进程。

表6-1 Windows Embedded CE 6.0中主要的系统进程

名 称	作 用
NK.exe	系统服务
Filesys.dll	对象存储
GWES.dll	图形、窗口和事件
Device.dll	设备管理
Explorer.dll	浏览器
Services.exe	服务管理
Repllog.exe	系统与外部设备连接
Rapisrv.exe	系统与Windows设备连接

在这里有一点需要指出的是，在Windows CE 5.0中，GWES里包含了电源管理功能，也就是说如果系统没有图形界面，Windows CE就不能进行相应的电源管理工作。而到了Windows Embedded CE 6.0中，电源管理功能被放在了NK.exe中，因此即使Windows CE设备没有包含GWES模块也可以实现对电源的管理。

6.1.2 创建进程

在Windows Embedded CE 6.0中，如果想创建进程，可以使用下面这个API函数：

```
BOOL CreatProcess(LPCTSTR lpApplicationName,
                  LPCTSTR lpCommandLine,
                  LPSECURITY_ATTRIBUSTES lpProcessAttributes,
                  LPSECURITY_ARRTIBUSTES lpThreadAttributes,
                  BOOL bInheritHandles, DWORD  dwCreationFlags,
                  LPVOID lpEnvironment,
                  LPCTSTR lpCurrentDirectory,
                  LPSTARTUPINFO lpStartupInfo,
                  LPPROCESS_INFORMATION lpProcessInformation);
```

因为Windows CE不支持安全性、继承性和当前目录，所以这个API函数的很多参数都必须设置成0或者False。因此上面这个函数就可以被简化成：

```
BOOL CreatProcess(LPCTSTR lpApplicationName,
```

```
        LPCTSTR lpCommandLine,NULL,NULL,FALSE,
        DWORD dwCreationFlags,NULL,NULL,NULL,
        LPPROCESS_INFORMATION  lpProcessInformation);
```

下面就来看看这个API函数中各个参数的含义：

- lpApplicationName：参数lpApplicationName为应用程序的名称。同Windows XP不同，这个参数不允许被设置成NULL。在Windows CE中，如果只传递了应用程序的名称而没有指定路径，那么系统将会优先搜索Windows目录，接着再搜索OEM厂商指定的路径。
- lpCommandLine：参数lpCommandLine用于传递启动参数。和Windows XP不同的是，表示参数的字符串在Windows CE中只能是UNICODE码。
- dwCreationFlags：参数dwCreationFlags为进程的状态标志，可以有下面几种选择：
  - 0：表示创建一个常规的进程。
  - CREATE_SUSPENDED：表示进程启动后被挂起。
  - DEBUG_PROCESS：表示创建的进程其功能是被父进程用来做调试。
  - DEBUG_ONLY_THIS_PROCESS：表示不能调试子进程。
  - CREATE_NEW_CONSOLE：表示当前进程为控制台程序。
- lpProcessInformation：这个参数表示传递一个LPPROCESS_INFORMATION结构的变量地址。

CreatProcess函数在创建进程成功的时候，会返回一个真值（True）；当创建进程失败的时候，将返回一个假值（False）。如果返回值为假，可以通过调用GetLastError函数来得到错误的信息。

其中，LPPROCESS_INFORMATION结构的定义形式为：

```
typedef struct _PROCESS_INFORMATION{
          HANDLE hProcess;
          HANDLE hThread;
          HANDLE dwProcessID;
          HANDLE dwThreadID;
}PROCESS_INFORMATION
```

这个结构中的前两个参数hProcess和hThread分别为新创建进程的句柄和新创建进程中主线程的句柄。dwProcessID和dwThreadID则为进程的ID号和进程中主线程的ID号。

刚才提到的Unicode是一种文字编码，不论使用什么平台的系统，什么样子的程序，什么类型的语言，Unicode都为每个字符提供了一个唯一的数字标示。Unicode标准已经被很多工业界的领导性企业所采用，如Apple、HP、IBM、Microsoft、Oracle、SAP、Sun、Sybase等。XML、Java等语言也都支持Unicode编码。

之所以使用Unicode编码，是因为计算机只能处理数字信息。计算机系统会指定一个数字来储存字母或其他字符。在Unicode被发明之前，有数百种标示这些字符的编码系统。但是没有一个编码可以包含足够的字符信息。即使单一语种的语言，如英语、意大利语等也没有哪一个编码可以适用于所有的字母、标点符号、常用的技术符号等。这些编码系统有时还会互相冲突。也就是说，不同的编码体系可能使用相同的数字代表两个不同的字符，或使用不同的数字代表相同的字符。任何一台特定的计算机，尤其是服务器都需要支持许多种不同的编码。尽管理论上不同种类的编码之间可以很好地进行转换，但是在真正转换的时候，总是难免会出现这样或者那样的错误。

下面以作者的姓“姜”为例加以说明。在简体中文（GB）、繁体中文（BIG5）中，尽管“姜”

都是同一个字，但是对应的编码却各不相同。在简体中文（GB）中，“姜”字的编码如果放在繁体中文（BIG5）中，可能会是另外一个字。但是如果采用Unicode来进行编码，则“姜”字只会有一个编码，而不管它在简体中文还是繁体中文中。

我们一定会有这样的体会，在早期的Office版本中，如果简单中文版本的Word拿英文版打开，就会显示乱码，这就是由于编码不同造成的。而最新版本的Office则采用了Unicode内核，因此无论何种文字都可以在上面正常地显示。

6.1.3 终止进程

在Windows Embedded CE 6.0中，终止一个进程最好是在WinMain函数中。当Windows CE中一个进程的主线程被终止时，不管其余线程是否处于激活状态，该进程都会被终止。在Windows CE中，如果想终止一个进程，那么首选的函数应该是ExitProcess()。这个函数的声明形式为：

```
VOID ExitProcess (UINT uExitCode);
```

里面唯一的参数uExitCode含义为退出代码。这个函数能够彻底地实现进程的关闭功能。也就是说，它将把进程终止的消息通知所有和该进程关联的动态链接库（DLL）。

可以通过下面的函数来获得指定进程的退出代码：

```
BOOL GetExitCodeProcess(HANDLE hProcess, LPDWORD lpExitCode);
```

当一个进程处于运行状态的时候，其退出代码为STILL_ACTIVE。当一个进程被成功终止的时候，其终止状态可能是下面的几种：

1）ExitProcess或者TerminateProcess函数指定的退出代码。

2）main或者WinMain函数中的返回值。

3）引起进程异常终止的代码。

例如，可以指定由内存不足导致的异常终止的退出代码为1，由文件读操作导致的异常终止的退出代码为2。这样就可以根据不同的退出代码进行各自不同的错误处理。

如果想在当前进程中终止另一个进程，则可以使用下面的这个TerminateProcess函数：

```
BOOL TerminateProcess(HANDLE hProcess, DWORD uExitCode);
```

其中，参数hProcess为进程的句柄，参数uExitCode为退出代码。如果终止进程的函数调用成功，则返回true，否则返回false。

需要注意的是，使用TerminateProcess函数会无条件地终止相关进程，该进程中的所有线程也会被终止。同时，还会把进程终止的消息通知给和这个进程有关的DLL文件。不过这个函数有很多的弊端（可能会导致处理进程结束工作的函数没有被执行完），一般不推荐使用。使用TerminateProcess函数终止进程，将会引起下面的这些变化：

1）被该进程打开的所有对象的句柄被关闭。

2）进程中的所有线程被停止执行。

3）进程的状态由STILL_ACTIVE变成相应的退出代码。

4）进程的状态变成等待态。

5）进程中线程的状态变成等待态。

6.1.4 其他相关函数

在终止一个进程的时候，必须首先知道相应进程的句柄。同时，在某些情况下也需要知道一些进程的句柄，因此这就需要有一个相应的函数来返回已经运行的进程的句柄。在Windows Embedded CE 6.0中，系统提供了这样一个函数。这个函数就是OpenProcess。

OpenProcess函数的函数原型为：

```
HANDLE OpenProcess (DWORD dwDesiredAccess, BOOL blnheritHandle,
                    DWORD dwProcessld);
```

下面来看看各个参数：

- dwDesiredAccess：在Windows CE中，系统不支持dwDesiredAccess参数，因此在这里要把该参数设置成0。
- blnheritHandle：在Windows CE中系统也不支持句柄继承，因此也需要把参数blnheritHandle设置为False。
- dwProcessld：参数dwProcessld表示想要打开进程的ID。

在Windows CE系统中，也可以使用另一个函数来获取窗口的句柄，并得到相应进程的ID值。这个函数就是GetWindowThreadProcessld函数。

GetWindowThreadProcessId函数的函数原型为：

```
GetWindowThreadProcessId (HWND hWnd, LPDWORD lpdwProcessId);
```

在Windows Embedded CE 6.0中，如果想直接读取一个进程的内存空间，那么可以使用ReadProcessMemory函数。这个函数的函数原型为：

```
ReadProcessMemory (HANDLE hProcess, LPCVOID lpBaseAddress,
                   LPVOID lpBuffer, DWORD nSize,
                   LPDWORD lpNumberOfBytesRead);
```

各个参数的含义为：

- hProcess：参数hProcess表示进程的句柄。
- lpBaseAddress：参数lpBaseAddress表示进程空间的基地址。
- lpBuffer：参数lpBuffer指定了将要读取数据的本地缓冲区的名称。
- nSize：参数nSize指定了将要读取数据的本地缓冲区的大小。
- pNumberOfBytesRead：参数pNumberOfBytesRead指定了读取的字节数。

如果想在Windows Embedded CE 6.0系统中直接写入一个进程的内存空间，则可以通过WriteProcessMemory函数。这个函数的函数原型为：

```
WriteProcessMemory (HANDLE hProcess, LPCVOID lpBaseAddress,
                    LPVOID lpBuffer, DWORD nSize,
                    LPDWORD lpNumberOfBytesWritten);
```

各个参数的含义为：

- hProcess：参数hProcess表示进程的句柄。
- lpBaseAddress：参数lpBaseAddress表示进程空间的基地址。
- lpBuffer：参数lpBuffer指定了将要写入数据的本地缓冲区的名称。
- nSize：参数nSize指定了将要写入数据的本地缓冲区的大小。

- pNumberOfBytesWritten：参数pNumberOfBytesWritten指定了写入的字节数。

6.2 Windows Embedded CE 6.0中的线程

从前面的介绍中可以看出，Windows Embedded CE 6.0和Windows XP中的每一个进程都会分成许多的线程。因此可以说，同进程相比，真正的执行单元实际上是线程。首先看看什么是线程。

6.2.1 线程概述

提到线程，首先就要回到20世纪60年代。那个时候，进程的概念刚刚被提出，进程就一直是操作系统中独立运行的最基本的单位。进程概念的提出，已经大大地提高了计算机资源的利用效率。但是在多个进程并发执行的时候，进程之间相互切换的开销却非常的严重，同时还影响到了进程之间通信的效率。于是，在20世纪80年代的中期，人们又提出了比进程更小的、可以独立运行的基本单位——线程。其目的就是减少程序并发执行时在时间和空间上的开销，进一步提高程序的并发性。

所谓线程，它实际上是进程的一个实体，是CPU调度和分配的基本单位。除了一些在运行中必不可少的资源（如寄存器和栈等）外，线程本身并不占有系统的资源。但是线程却可以和同属于一个进程的另一个线程共享进程所拥有的全部资源。同进程一样，一个线程可以创建或者撤销另一个线程，同一个进程内的线程也可以并发执行。

Windows Embedded CE 6.0中的线程和Windows XP中的线程比较类似，每一个进程中都包含一个主线程。通过一定的函数，在一个进程中可以创建许许多多的线程，在Windows CE中创建线程的数量只受内存空间大小和线程栈空间大小的影响。进程中的线程会占用进程的地址空间，各个线程之间也拥有同样的访问权限。

在一些书籍中有时也会把线程称为轻量级进程，这是因为线程和进程具有许多相似的地方。下面就来看一下两者之间到底有什么样的区别和联系。

1）从并发性的角度来看，引入的线程不仅可以在不同的进程之间并发执行，在同一个进程中的多个线程之间也是可以并发执行的。因此这种方式可以使操作系统具有更好的并发执行能力，可以更为有效地利用系统资源，同时在一定程度上极大地提高了系统的运行效率。

2）从系统的开销来看，进程在创建或者被终止的时候，系统需要为进程分配或者回收资源，而线程之间的转换只需要保存和设置少量的寄存器内容，因此进程的开销比较大，而线程的开销则比较小。

3）从调度方面看，本章前面提到的进程在早期的操作系统中是分配资源、调度和分配的基本单位。而在引入线程以后，线程则代替进程成为了调度和分配的基本单位。但是拥有资源的基本单位仍旧是进程。这就使得进程可以轻快地运行，提高了系统的并发程序。

4）从占有的资源来看，进程是占有资源的最基本的单位。一般地，线程不占有资源，但是它可以访问其所属进程的资源。

尽管进程和线程有很多类似的地方，但是作为一个新引入的概念，线程自然也具有其独特的一些属性。这些特性主要体现在：

1）两个线程之间的切换所花费的时间和进程切换相比，要少得多。

2）创建一个线程所花费的时间和创建一个进程多花费的时间相比，也要少得很多。

3）因为创建进程中的线程不要额外地分配资源，因此创建线程的速度要比创建进程快得多。

4）同样因为创建线程不需要额外的资源，其创建时候的系统开销也非常的小。

5）线程之间的通信是不需要额外的机制来保障的，因为同一个进程内的线程共享其拥有的资源，这样也就使得线程之间的通信速度要比进程之间快得多。

在早期的Windows CE版本中，系统只支持8个级别的优先级调度。从Windows CE 3.0开始，增长到了256个优先级。在这些优先级中，0优先级级别最高，255优先级级别最低。0~247的优先级属于实时性优先级，而248~255的优先级一般分配给普通应用程序来使用。具体的优先级范围为：

- 0~96：优先级高于驱动程序的程序。
- 97~152：基于Windows CE.NET的驱动程序。
- 153~247：优先级低于驱动程序的程序。
- 248~255：普通的应用程序。

尽管如此，大多数的应用程序仍旧使用最基本的8个优先级。剩下的248个优先级大多被驱动程序所使用。如果一个应用程序需要使用高一些的248个优先级，则需要特殊的API函数进行调用。最低级别的8个优先级分别为：

- THREAD_PRIORITY_TIME_CRITICAL：表示比正常的级别高3个等级。
- THREAD_PRIORITY_HIGHEST：表示比正常的级别高2个等级。
- THREAD_PRIORITY_ABOVE_NORMAL：表示比正常的级别高1个等级。
- THREAD_PRIORITY_NORMAL：表示正常的级别。所有被创建的线程默认为这个 级别。
- THREAD_PRIORITY_BELOW_NORMAL：表示比正常的级别低1个等级。
- THREAD_PRIORITY_LOWEST：表示比正常的级别低2个等级。
- THREAD_PRIORITY_ABOVE_IDLE：表示比正常的级别低3个等级。
- THREAD_PRIORITY_IDLE：表示比正常的级别低4个等级。

根据优先级大小的不同，所有高优先级的线程都运行在低优先级的线程之前。所有低优先级的线程不能抢占高优先级的线程。也就是说，在一个设置为某个优先级别的线程能够调用之前所有具有较高优先级别的线程被阻塞（被阻塞的线程指的是那些还没有用完自己的时间片，但是用于等待同步对象或者等待某种系统资源的线程）。而具有相同优先级别的线程则采用时间片轮转的方法来调度。一旦一个线程自动放弃它的时间片，或者用完了它的时间片，它将被阻塞。这样，其余相同优先级别的线程就可以在这个线程继续运行之前开始运行。

如果系统中一个高优先级别的线程被清除了阻塞状态，而正好有一个低优先级别的线程处在运行中，那么这个低优先级别的线程将被挂起，系统转而运行这个高优先级别的线程。这样，高优先级别的线程就抢占了低优先级别线程的运行权。

但是有两种情况比较例外。一种是一个线程具有了THREAD_PRIORITY_TIME_CRITICAL优先级别，这个线程将会导致所有线程的等待，直到它运行结束为止。正是具有了这种特性，THREAD_PRIORITY_TIME_CRITICAL这个优先级一般被保留，用来在设备驱动程序的编程中使用。

另外，当一个低优先级别的线程拥有运行时所必须的一些资源，而高优先级别的线程在等待这个资源，那么这个低优先级别的线程将被临时赋予较高的优先级别，以便保证它可以在运行完以后，释放掉所占有的资源。

在Windows CE中，线程一般分为5种状态，分别为运行状态（running）、挂起状态（suspended）、睡眠状态（sleeping）、阻塞状态（blocking）和终止状态（terminated）。其中，当系统内所有的线程处于阻塞状态的时候，系统的内核则处于空闲模式（Idle Mode），此时对CPU的电力资源的占用将减少。要知道，对于大多数靠电池供应电力的Windows CE设备（如Pocket PC、Smart Phone等）来说，对电力资源占用的大小是至关重要的。

6.2.2 创建线程

在Windows Embedded CE 6.0中，可以通过CreateThread函数来创建线程，其函数原型为：

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
                    DWORD dwStackSize,
                    LPTHREAD_START_ROUTINE lpStartAddress,
                    LPVOID lpParameter, DWORD dwCreationFlags,
                    LPDWORD lpThreadID);
```

关于这个函数，有几点需要特别说明：

1）在Windows CE中，由于CreateThread函数不支持个别的属性，因此有些属性参数需要被设置成0或者NULL。如Windows CE.NET中不支持安全性属性，因此参数lpThreadAttributes需要被设置成0。

2）当参数dwCreationFlags被设置成STACK_SIZE_PARAM_IS_A_PESERVATION的时候，参数dwStackSize就可以指定初始化时栈的大小，此时Windows CE系统的内核将按照此参数设置的数值来保留栈的空间地址。

3）当参数dwCreationFlags没有被设置成STACK_SIZE_PARAM_IS_A_PESERVATION的时候，参数dwStackSize必须被设置成0，此时表示按照默认数值进行栈空间大小的设置。

4）当参数dwCreationFlags被设置成CREATE_SUSPENDED的时候，表示被创建的线程在创建以后一直处于挂起状态，直到ResumeThread函数将该线程恢复运行为止。

当这个函数创建线程成功以后，将会返回新创建线程的句柄。当撤销线程的时候，则需要释放掉这个线程的句柄。

6.2.3 设置线程的优先级

在前面已经提到，被创建的线程一般默认被设置成THREAD_PRIORITY_NORMAL。那么如何更改线程的优先级呢？可以通过下面这个函数：

```
BOOL SetThreadPriority (HANDLE hThread, int nPriority);
```

各个参数的含义为：

- hThread：参数hThread为线程的句柄。
- nPriority：参数nPriority为新的优先级别。这个优先级别可以为前面提到的THREAD_PRIORITY_TIME_CRITICAL到THREAD_PRIORITY_IDLE。

在这里需要注意的是，在更改线程优先级别的时候要特别的小心，防止出现更改优先级以后的线程影响到已经存在的其余线程。因为一般情况下，高优先级别的线程在低优先级别的线程之前被执行。

更改线程优先级别的另外一个函数为：

```
BOOL CeSetThreadPriority (HANDLE  hThread, int  nPriority);
```

和前面那个函数一样，两个参数的含义分别为线程的句柄和新的优先级别。那么这两个设置线程优先级别的函数有什么不同呢？下面就来看一下。

函数CeSetThreadPriority可以把线程设置成所有256个优先级中的任意一种。其中0代表优先级别最高，255代表优先级别最低。而函数SetThreadPriority只能把线程设置成最基本的8个优先级别。同时，只有被信任的线程、安全级别很高的线程才能使用CeSetThreadPriority函数更改线程的优先级别。

6.2.4 查询线程的优先级别

查询线程的优先级别可以通过下面这个函数：

```
int GetThreadPriority (HANDLE hThread);
```

这个函数实际上是和函数SetThreadPriority相对应的。当线程的优先级别高于THREAD_PRIORITY_TIME_CRITICAL的时候，该函数返回的是和THREAD_PRIORITY_TIME_CRITICAL所对应的数值。

因此，当需要查询优先级别比THREAD_PRIORITY_TIME_CRITICAL高的线程的时候，可以通过下面这个函数：

```
int CeGetThreadPriority (HANDLE hThread);
```

这个函数和函数CeSetThreadPriority相对应，它的返回值为0~255之间的任意一个数值，其中0代表优先级别最高。需要注意的是，在Windows CE中，系统采用了不同的数值来表示两个函数的返回结果。例如，对于THREAD_PRIORITY_NORMAL级别的线程，函数GetThreadPriority的返回值是3，而CeGetThreadPriority的返回值则是251。

6.2.5 设置线程的时间片

在Windows CE中，线程可以设置各自的时间片（time quantum）。时间片指的是内核限制线程占用的CPU的时间，其默认值为100ms。OEM厂商可以修改这个值，如对于大多数的Pocket PC设备来说，其线程时间片为75ms，有的设备也被设置成了100ms。

可以通过下面的函数来设置线程的时间片：

```
int CeSetThreadQuantum (HANDLE hThread, DWORD dwTime);
```

第一个参数为线程的句柄，第二个参数为设置的时间片的值。如果把时间片设置成0，该线程就会一直占用CPU的时间，直到有更高优先级别的线程要求占用CPU时间或者当前线程运行结束为止。

如果想查询一个线程的时间片，则可以通过下面这个函数：

```
int CeGetThreadQuantum (HANDLE hThread);
```

和前面的函数一样，里面的参数为线程的句柄。这个函数执行完以后将返回该线程的时间片。

6.2.6 挂起和恢复一个线程

如果想在程序中挂起一个线程，则可以通过下面这个函数：

```
DWORD SuspendThread (HANDLE hThread);
```

里面的参数为将要被挂起的线程的句柄。如果这个函数被调用成功，指定的线程将被挂起，并且线程挂起的次数递增。它的返回值是线程被挂起的次数。

如果想恢复一个线程，则用下面的函数：

```
DWORD ResumThread (HANDLE hThread);
```

同样，里面的参数为线程的句柄。如果函数调用成功，将返回挂起的次数。每一次这个函数被成功执行以后，线程的挂起次数将递减。当函数的返回值为0 的时候，指定的线程将恢复运行。

在这里需要注意的是，在调用SuspendThread函数的时候，如果线程正在执行一个系统调用，那么挂起此线程就会失败。因此为了达到挂起线程的目的，可以多次调用SuspendThread函数，直到线程被挂起成功。

6.3 进程间通信

在进行程序设计的时候，会经常遇到进程同步的情况。因为在大多数情况下，进程之间要相互通信、相互协调才能完成指定的任务。例如，当有很多个线程访问一个系统资源的时候，就必须保证其在访问资源时，其他的线程不能够修改这个系统资源。只有这样才能保证数据的一致性。再如，有两个线程，其中一个进程在准备执行下一个任务之前，它必须等待另一个进程终止任务以后才能运行，这种情况也需要两者之间的互相协调。

同桌面PC系统相比，Windows CE系统中没有提供消息、端口等复杂的通信机制，而是提供了用于等待队列的几种基本的通信手段，如事件对象、互斥、临界区等。这些都被以系统对象的形式来实现。

所谓系统对象，就是由系统管理的资源实体，系统通过列表记录并跟踪它们的状态。当某一个进程向系统请求一个系统对象的时候，系统将返回一个用来标示对象的句柄。这样，进程就可以通过这个句柄访问该对象。这些保证进程间通信的手段对于嵌入式系统来说，方法简单实用，并且可以以较小的系统开销，实现较为完善的功能。

下面就来具体看一下在Windows Embedded CE 6.0中各种进程间的通信手段。

6.3.1 事件对象

事件对象（Events）是同步对象之一，一个事件对象就类似一种特定的短消息，它被用来通知某个线程发生了一个特定的事件。有的时候也会告诉这个线程应该做什么事情了。例如，当系统的即插即用管理器发现了一个新插入的设备的时候，它就会设定一个特定的事件，这样系统就会触发和该事件相关的线程，并告诉它系统已经添加了一个新的设备。

事件对象有信号（signaled）和无信号（nonsignaled）两种状态。事件对象在创建的时候可以选择自动地从signaled状态恢复到nonsignaled状态，或者手动恢复到这个状态。事件对象是可以被用于共享进程间的同步通信的。

可以通过下面这个函数来创建事件对象：

```
HANDLE CreateEvent (LPSECURITY_ARRRIBUTES, lpEventAttributes,
                    BOOL bManualReset, BOOL bInitialState,
                    LPTSTR lpName);
```

各个参数的含义为：

- lpEventAttributes：参数lpEventAttributes表示安全属性，和前面介绍的函数一样，在Windows CE中，这个参数应该被设置成NULL。
- bManualReset：参数bManualReset表示是否手动设置事件对象的状态。这个值如果为false，表示等待函数接到信号以后，返回的此次事件对象被自动地设置成无信号状态；如果其值被设置成true，则表示等待函数接到信号以后，返回的事件不会被自动地设置成无信号状态。
- bInitialState：参数bInitialState表示事件对象初始化的时候是否被设置成信号状态或者无信号状态。
- lpName：参数lpName表示事件对象的名字。当两个进程采用了同一个名字以后，进程就可以共享同一个对象。如果不想设置事件对象的名称，只需要把这个参数设置成NULL即可。

需要注意的是，如果想在进程之间共享一个事件对象，那么就需要在每一个进程里面创建一个事件对象，而不能只在一个里面创建，然后把事件对象的句柄传递给另一个进程。

如果想知道通过CreateEvent函数创建的事件对象是新创建的，还是仅仅打开了一个已经存在的事件对象，可以在这个函数的后面添加上GetLastError函数。如果这个函数返回ERROR_ALREADY_EXISTS，那么表示打开的是一个已经存在的事件对象。

再来看看下面这两个函数：

```
BOOL SetEvent(HANDLE hEvent);
BOOL PulseEvent(HANDLE hEvent);
BOOL ResetEvent(HANDLE hEvent);
```

函数SetEvent()的作用为将事件对象设置为有信号状态。PulseEvent()的作用为将事件对象设置成有信号状态以后，立即设置为无信号状态。如果想手动地把事件对象设置成无信号状态，则需要使用ResetEvent()函数。

6.3.2 线程等待

在Windows CE中，与线程等待相关的函数主要有下面几个：

- WaitForSingleObject。
- WaitForMultipleObjects。
- MsgWaitForMultipleObjects。
- MsgWaitForMultipleObjectsEx。

其中WaitForSingleObject的函数原型为：

```
DOWRD WaitForSingleObject (HANDLE hHandle, DWORD dwMilliseconds);
```

此函数仅当在参数列表中指定的对象处于信号态或超过了超时间隔时，才返回。函数中的参数hHandle为标识的同步对象，其等待的时间为dwMilliseconds指定的值，等待时间的单位为毫秒。当给参数dwMilliseconds传递的值为INFINITE的时候，表示该对象要无限期地等待下去。这个函数的返回值可以为：

- WAIT_OBJECT_0：表示同步的对象成为了信号状态。
- WAIT_TIMEOUT：表示同步的对象在指定的时间内没有成为有信号状态，也就是说在超时范围内，期望的事件没有发生。
- WAIT_ABANDONED：表示拥有互斥对象的线程被终止。
- WAIT_FAILED：表示这个函数调用失败。

在Windows CE中，一个线程可以等待许多个事件，当其中的任何一个事件被标识为信号状态的时候，线程的等待将中止。要想实现这个功能，可以使用下面这个函数：

```
DWORD WaitForMultipleObjects (DWORD nCount, CONST HANDLE *lpHandles,
                              BOOL fWaitAll, DWORD dwMilliseconds)
```

其各个参数的含义为：

- nCount：参数nCount指定了同步对象的数量。
- *lpHandles：*lpHandles指向包含的所有同步对象。
- fWaitAll：当参数fWaitAll被设置成FALSE的时候，表示只要有一个同步对象处于信号状态，那么线程就被唤醒而继续运行；如果被设置成TRUE，则表示函数要等待所有同步对象全部处于信号状态的时候，线程才被唤醒。
- dwMilliseconds：dwMilliseconds指定了等待的时间。

这个函数返回的是同步对象在数组中的索引。其中WAIT_OBJECT_代表第一个元素；WAIT_OBJECT_0+n代表第（n+1）个元素。如果在指定的时间内没有任何同步的对象处于信号状态，那么该函数返回WAIT_TIMEOUT。如果此函数调用失败，则函数返回值为WAIT_FAILED。

下面看看MsgWaitForMultipleObjects函数，其函数原型为：

```
DWORD MsgWaitForMultipleObjects (DWORD nCount, LPHANDLE pHandles,
                                 BOOL fWaitAll,
                                 DWORD dwMilliseconds,
                                 DWORD dwWakeMask)
```

这个函数的功能同WaitForMultipleObjects函数类似，只是增加了一个和消息相关的、被称为唤醒掩码（dwWakeMask）的参数。这个函数不但能够等待内核对象，还可以等待指定的消息。

再看看下面这个函数：

```
DWORD MsgWaitForMultipleObjectsEx (DWORD nCount, LPHANDLE pHandles,
                                   BOOL fWaitAll, DWORD dwMilliseconds,
                                   DWORD dwWakeMasks);
```

这个函数实际上是MsgWaitForMultipleObjects的扩展，如果一个线程在执行大量任务的同时还要响应用户的按键消息，MsgWaitForMultipleObjects函数和MsgWaitForMultipleObjectsEx函数将会起到非常大的作用。在Windows CE中，系统只支持下面几种类型的dwWakeMasks标示：

- QS_ALLINPUT：表示收到了消息。
- QS_INPUT：表示收到了一个输入消息。
- QS_KEY：表示收到了一个按键按下、释放的消息。
- QS_MOUSE：表示收到了一个鼠标移动或者单击的消息。

- QS_MOUSEBUTTON：表示收到了一个鼠标单击的消息。
- QS_MOUSEMOVE：表示收到了一个鼠标移动的消息。
- QS_PAINT：表示收到了一个WM_PAINT消息。
- QS_POSTMESSAGE：表示收到了一个POST消息。
- QS_SENDMESSAGE：表示收到了一个发送消息。
- QS_TIMER：表示收到了一个WM_TIMER消息。

6.3.3 信号量

信号量机制是一种非常有效的进程同步机制，它已经被广泛地应用于各种类型的操作系统中。从前面的介绍中可以知道，事件对象（Event Object）可以处于信号状态（signaled）和非信号状态（nonsignaled）。同样，当可用资源大于0的时候，信号对象处于有信号状态；当可用资源等于0的时候，信号对象处于非信号状态。

在Windows CE中，当线程在等待信号量的时候，该线程处于锁住（blocked）状态，这种状态一直持续到可用资源大于0为止。可用资源的最大值是在创建信号量的时候被指定的，因此可以通过信号量来指定访问资源的线程的数量。例如，当系统中有一个10个数据大小的缓冲区的时候，可以允许最多10个线程同时访问。当第11个线程需要访问这个被信号量保护的缓冲区的时候，这个线程将被锁住，直到其中的一个线程释放掉它的信号量为止。

可以通过下面的函数来创建一个信号量：

```
HANDLE CreateSemaphore (LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
                        LONG lInitialCount, LONG lMaximumCount,
                        LPCTSTR lpName);
```

各个参数的含义为：

- lpSemaphoreAttributes：同前面介绍的函数一样，在Windows CE中，lpSemaphoreAttributes被设置成NULL。
- lInitialCount：参数lInitialCount表示当前可用资源的初始值。
- lMaximumCount：参数lMaximumCount为最大可用资源数量。
- lpName：参数lpName为对象的名字。

当参数lInitialCount的值等于0的时候，信号对象处于非信号状态，此时内核将调用等待函数的线程设置成睡眠状态；当lInitialCount的值大于0的时候，则信号对象处于信号状态，这个时候内核将调用等待函数的线程设置成运行状态，并将信号对象当前可用的资源数减1。

需要注意的是，如果两个线程通过CreateSemaphore函数创建的信号量的名称相同，那么第二个线程的CreateSemaphore函数将不创建新的信号量对象，而是返回一个指向第一个线程创建的信号量的句柄。

当线程访问完可用资源以后，将要释放掉信号量。释放信号量的函数为：

```
BOOL ReleaseSemaphor (HANDLE hSemaphore, LONG lReleaseCount,
                      LPLONG lpPreviousCount);
```

各个参数的含义为：

- hSemaphore：参数hSemaphore为信号量对象的句柄。

- lReleaseCount：参数lReleaseCount为要释放的资源数，这个参数的值必须大于0。
- lpPreviousCount：参数lpPreviousCount返回原来的可用资源数。如果不需要知道原来的可用资源数，则可以把这个参数设置成NULL。

当一个线程访问完可用资源以后，必须使用ReleaseSemaphor函数释放信号量，以使当前可用资源数递增。同时在Windows CE下，系统不支持OpenSemaphore函数。还有一点需要强调，等待函数在成功等待信号量对象后会把当前可用资源数减1，但是因为线程可以一次使用多个资源，因此这就有可能导致问题的出现。为了避免上面提到的问题，应该遵循一个线程一次只使用一个资源的原则。

6.3.4 互斥

互斥也是一种保证线程同步的手段之一，它能够保证多个线程对同一共享资源的互斥访问。只有拥有互斥对象的线程才具有访问资源的权限。同时，由于互斥对象只有一个，因此就决定了任何情况下共享资源都不会同时被多个线程所访问。同前面的信号量一样，占有资源的线程在处理完任务以后要将拥有的互斥对象交出，以保障其他线程在获得互斥对象以后可以访问资源。图6-1给出了互斥对象的工作模型。

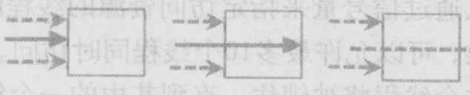


图6-1 拥有互斥对象的线程访问共享资源

图6-1中的方块代表共享资源，实线箭头代表拥有互斥对象的线程，虚线箭头代表不拥有互斥对象的线程。首先来看一下图6-1a表示的情况。此时，有3个线程需要访问共享资源，但是只有中间的一个线程拥有互斥对象，这样只有这个线程可以访问到共享资源，而其余的线程则被阻挡在外面，此时的情况如图6-1b所示。当拥有互斥对象的进程访问完资源以后，会释放掉拥有的互斥对象，以便其余得到互斥对象的线程可以继续访问，这一过程如图6-1c所示。

如果想创建一个互斥对象，可以通过下面这个函数来实现：

```
HANDLE CreateMutex (LPSECURITY_ATTIBUTES lpMutexAttributes,
                    BOOL bInitialOwner, LPCTSTR lpName);
```

各个参数的含义为：

- lpMutexAttributes：和前面一样，参数lpMutexAttributes必须被设置成NULL。
- bInitialOwner：参数bInitialOwner如果被设置成True，则表示当前线程占有互斥资源，互斥对象的线程ID被设置成当前线程的ID，递归计数器被设置为1，互斥对象处于无信号状态。当bInitialOwner被设置成False的时候，则表示当前线程不占有互斥资源，互斥对象包含的线程ID和递归计数器都被设置为0。
- lpName：参数lpName为互斥对象的名称。

当互斥对象被创建一个，如果想获得这个互斥对象，首先应该通过GetLastError函数来验证互斥资源是否已经被提交，如果已经被前一个使用它的线程提交，那么就可以通过WaitForSingleObject函数来获得这个互斥对象。

如果想释放掉一个互斥对象，则可以通过下面这个函数来实现：

```
BOOL ReleaseMutex (HANDLE hMutex);
```

里面的参数表示互斥对象的句柄。

当一个线程在已经拥有互斥对象的同时，又调用等待函数等待同一个互斥对象，等待函数会立即返回相应的值，因为这个线程已经占有了互斥对象。当一个线程被唤醒并且在访问资源资源结束以后，必须调用ReleaseMutex函数，将互斥对象的递归计数器减1。调用ReleaseMutex函数的次数必须和调用等待函数的次数相同。和Windows XP不同的是，在Windows CE中，不支持OpenMutex函数。

6.3.5　互锁函数

在Windows CE中，互锁函数的作用是保证当一个线程访问一个变量的时候，其他线程无法访问此变量，以确保变量值的唯一性，这种访问方式称为原子访问。在Windows CE中，系统支持完整的互锁Win32 API函数。常见的互锁函数主要有：

```
LONG InterlockedIncrement (LPLONG lpAddend);
```

这个函数使一个LONG类型的变量增加1，里面的参数为指向这个变化变量的指针。

```
LONG InterlockedDecrement (LPLONG lpAddend);
```

这个函数使一个LONG类型的变量减少1，里面的参数为指向这个变化变量的指针。

```
LONG InterlockedExchange(LPLONG Target, LONG Value);
```

这个函数的作用是将参数Value的值赋给Target指向的值，函数将返回参数Target指向变量的初始值。

```
LONG InterlockedCompareExchange(LPLONG Destination, LONG Exchange,
                                LONG Comperand);
```

这个函数的作用是将参数Destination指向的值和参数Comperand的值进行比较，如果这两个值相同，则把参数Exchange的值赋给参数Destination指向的值；如果不相同，则不变。函数返回的是参数1指向的变量的初始值。

```
LONG InterlockedTestExchange (LPLONG target, LONG OldValue,
                              LONG NewValue);
```

这个函数的作用是将参数target指向的值和OldValue的值进行比较，如果这两个值相同，则把参数NewValue的值赋给target指向的值；如果不相同，则不作变化。

```
LONG InterlockedExchangedAdd(LPLONG Addend, LONG Increment);
```

这个函数的作用是将参数Increment的值加到参数Addend指向的值中，函数返回的是参数Addend指向变量的初始值。

```
LONG InterlockedCompareExchangePointer (PVOID *Destination, PVOID Exchange,
                                        PVOID Comperand);
```

这个函数的作用和函数InterlockedCompareExchange相同，只不过传递的数据都是指针类型的。函数的返回值为参数1指向指针的初始值。

```
LONG InterlockedExchangePointer (PVOID * Target, PVOID Value);
```

这个函数的作用是同函数InterlockedExchange相同，只不过传递的数据是指针类型的。函数的返回值仍旧为指针1指向指针的初始值。

6.3.6 临界区

临界区是Windows CE系统内部最常用的互斥手段。它能保证在临界区内所有被访问的资源不被其他线程访问，直到当前线程执行完临界区的代码。一般情况下，临界区对象用于保证一段程序代码的不间断性。一个临界区对象的使用通常被限定在某一个进程或者动态链接库中。

与临界区相关的函数主要有：

```
void InitializeCriticalSection (LPCRITICAL_SECTION);
void EnterCriticalSection (LPCRITICAL_SECTION);
void LeaveCriticalSection (LPCRITICAL_SECTION);
void DeleteCriticalSection (LPCRITICAL_SECTION);
BOOL TryEnterCriticalSection (LPCRITICAL_SECTION);
```

这些函数的主要功能为：

- InitializeCriticalSection：函数InitializeCriticalSection的作用是初始化临界区。
- EnterCriticalSection：函数EnterCriticalSection的作用是进入临界区。
- LeaveCriticalSection：函数LeaveCriticalSection的作用是退出临界区。
- DeleteCriticalSection：函数DeleteCriticalSection的作用是撤销临界区。
- TryEnterCriticalSection：函数TryEnterCriticalSection的作用是尝试进入临界区。

上面列举的5个函数中都只有一个参数，就是LPCRITICAL_SECTION结构体变量。LPCRITICAL_SECTION结构是在WINBASE.H中定义的，应用程序不需要对这个结构中的任何字段进行操作。

通过使用InitializeCriticalSection函数可以向系统申请一个临界区对象，在使用完以后，需要调用DeleteCriticalSection函数来释放这个临界区对象。同时，还要保证以EnterCriticalSection函数或者TryEnterCriticalSection函数开始，以LeaveCriticalSection函数结束的代码段中，临界区代码执行时候的相关临界区句柄必须是有效的。这样，系统就可以保证当多个线程尝试进入到同一段临界区代码的时候，只会有一个是成功的。

从上面的介绍中可以了解到：

- 当创建临界区的时候，需要使用InitializeCriticalSection函数。
- 当线程进入到一个受保护的代码区域的时候，需要使用EnterCriticalSection函数。在这里有一点需要注意的是，该函数是把LPCRITICAL_SECTION结构作为自己的唯一参数，同时LPCRITICAL_SECTION结构是用InitializeCriticalSection函数来初始化的。
- 当线程离开临界区的时候，需要使用LeaveCriticalSection函数。在这里有一点需要注意的是，因为临界区需要跟踪使用的次数，因此每当调用一次EnterCriticalSection函数，就需要调用一次LeaveCriticalSection函数，通俗地讲，两个函数要“配对”。
- 当不再使用临界区的时候，需要调用DeleteCriticalSection函数，此时将会清除用来管理临界区的所有系统资源。

第7章 Windows Embedded CE 6.0中的文件系统和注册表管理

7.1 Windows Embedded CE 6.0中的文件系统

在介绍完Windows Embedded CE 6.0中的内存管理、进程、线程等知识以后，接下来介绍的是Windows CE 6.0中的文件系统。

从前面的介绍中可以知道，Windows CE和传统的Windows系统非常类似。无论用户界面、常用的应用软件，还是其内部的运行机制等。但是，Windows CE的文件系统却和目前被广泛使用的Windows XP有着非常大的不同。

在Windows CE中，系统采用了一种不同于桌面PC那样以硬盘为存储介质的存储方式，而是实现了一种特别的、基于ARM的文件系统，称为对象存储。系统采用对象存储方式在存储文件的同时，也存储着Windows CE的注册表和系统的数据库。

Windows CE中的文件API函数和前面几章介绍的内容一样，直接来源于Win32。尽管Windows CE已经实现了标准的注册表API，但是和传统的Windows相比，仍旧无法实现那样高的安全机制，一些函数中表示安全权限的参数在Windows CE中并不被支持，因此这些参数经常被设置成NULL，关于这一点可以从有关的函数介绍中体会到。

概括地讲，Windows Embedded CE 6.0在存储管理和文件系统方面提供的支持主要表现在：

- FAT文件系统。
- 对象存储。
- 嵌入式数据库。
- 注册表。
- 第三方程序的加密、解密。

首先来看一下Windows Embedded CE 6.0中的文件系统。

7.1.1 文件系统概述

在Windows Embedded CE 6.0中，系统支持两种类型的文件系统，它们分别是：

- 文件系统驱动（File System Drivers）控制的文件系统。
- 注册文件系统（Registered File System）。

其中文件系统驱动控制的文件系统是Windows Embedded CE 6.0首选的文件系统，它包含了大量的文件系统驱动程序。而注册文件系统又包含三类，分别为：

- 基于RAM的文件系统。
- 基于ROM的文件系统。

- RELFSD（Release-Directory File System）。

其中，RAM文件系统和ROM文件系统是Windows Embedded CE 6.0的内置文件系统，如Windows CE的系统程序等都被安装到ROM中等。剩下的一种则属于可安装文件系统。另外，嵌入式系统的开发人员也可以编写自己的文件系统并在系统中注册使用。在Windows CE设备中，无论存储设备属于何种类型，所有对文件系统的访问都是通过Win32 API完成的。因此也可以说，Windows CE提供了设备无关的文件访问。

同时，在Windows CE平台中，RAM存储器是其默认使用的存储设备，被称为对象存储(object store)。需要注意的是，Windows CE中的RAM存储器与PC上使用的RAM不同，它在关机的状态下仍旧有电流供应，因此即使关闭机器，里面存储的数据也不会丢失。刚刚提到的对象存储在Windows CE中所起到的作用就相当于PC上的硬盘，它为嵌入式设备中的应用程序和数据提供了相对稳定和持久的存储环境。除了用于存放文件以外，对象存储中还可以存储Windows CE系统的数据库和系统的注册表。

下面就来看一下什么是Windows Embedded CE 6.0系统中的对象存储。

7.1.2 对象存储

在Windows Embedded CE 6.0中，系统默认使用的文件系统就是对象存储。对象存储在Windows CE中的作用和PC上的硬盘非常相似，它为应用程序以及相关数据提供了相对持久稳固的存储。只要系统中的电池有电量供应，对象存储中保存的信息就不会丢失。在使用嵌入式设备的过程中，很难分辨出存储在RAM中的对象存储文件和存储在ROM中的文件有什么不同，存储在两者中的文件也是被放在同一个目录下的。在Windows CE系统中，最多可以支持256个不同的存储设备。这些设备的接口就是可安装文件系统（Installable File System，IFS）API。许多Windows CE系统平台都包含了FAT文件系统的IFS驱动程序，第三方的嵌入式设备制造厂商也可以编写自己的IFS驱动程序，用以支持不同的文件系统。

在Windows CE中，对象存储实际上包含了三个部分，它们分别是：

- 文件系统。
- 数据库。
- 系统注册表。

这三个部分共享一个内存堆。但是，文件系统、数据库和系统注册表不一定要驻留在对象存储中，它们可以被存放在ROM或者外部的存储设备中。Windows CE中的对象存储是由FileSys.exe文件控制的。同时系统中数据的创建和使用与实际的存储设备无关，只是依赖于存储设备的类型。

在Windows CE系统中，操作系统主要负责管理内存堆，在需要的时候才对文件进行解压和压缩，同时它还提供了基于ROM的应用程序和基于RAM的数据间的无缝集成。如果在对对象存储写入数据的时候发生电源中断的情况，Windows CE系统则会通过各种方式来保证对象存储不被破坏。如可以采取系统重新启动以后继续完成该操作，或者恢复到电源中断前的存储状态。

在前面的章节中也提到过，在早期的Windows CE版本中，RAM文件系统和单个文件的大小最大只能是16MB。发展到Windows CE 3.0版本的时候，RAM文件系统最大已经达到了256MB，而单个文件系统的最大值也变成了32MB。同时，对象存储中的对象数目也由原来的65 536增长

到了将近4 000 000个。

在Windows CE中，系统为对象存储中的每一个对象都分配了一个唯一的对象标识符，该标识符主要用于访问对象存储中的对象。在Windows CE的对象存储中，以下各项都可以被定义为一个对象：

1）注册表中的1个值；

2）注册表中的1个键；

3）一个数据库；

4）数据库中的一条记录；

5）数据库中一条记录的扩展信息；

6）数据库中的一个卷；

7）一个文件。

在这里有一点需要特别地说明，Windows CE系统仅仅保证了对象标识符在数据库中的一个卷中是唯一的。当在数据库中有很多卷的时候就不能保证其唯一性了。这个时候，Windows CE系统就为每个数据库分配了一个全球唯一的标识。这样，只需要将这个全球唯一标识和前面提到的唯一对象标识符组合起来使用，就可以唯一地确定对象。

7.1.3 文件的输入/输出

同Window XP一样，Windows CE也支持许多类似的文件输入/输出函数，如CreateFile、ReadFile、WriteFile、CloseFile函数等。但是，作为一个独具特色的嵌入式操作系统，也存在着一些微小的差别。例如，不支持Win16标准下的_lread、_lwrite、_llseek；不支持用FILE_FLAG_OVERLAPPED标识打开设备或者文件等。这些问题在Windows CE系统中都可以以别的方式加以解决。

Windows CE系统中的文件操作和MS-DOS一样，都是采用基于句柄的操作方式。例如，当通过一个函数打开文件的时候，函数将返回相应文件的句柄，使用读写函数的时候，相应函数的句柄会指向需要操作的文件；还可以通过文件指针的偏移来进行文件数据的读写操作；当对文件的操作结束以后，只需要关闭文件的句柄即可。

下面就来具体地看一下这些函数。

1. 创建和打开文件

创建或者打开文件的函数为：

```
HANDLE CreateFile (LPCTSTR lpFileName,
                   DWORD dwDesiredAccess,
                   DWORD dwSharedMode,
                   LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                   DWORD dwCreationDistribution,
                   DWORD  dwFlagsAndAttributes,
                   HANDLE hTemplateFile);
```

各个参数的含义为：

- lpFileName：参数lpFileName为将要被创建的或者将要被打开的文件的名称；文件名中需要包含完整的文件路径。如果不包含路径信息，系统将默认为对象存储目录中的根目录。

- dwDesiredAccess：参数dwDesiredAccess的含义为文件的访问权限。其中：
  - GENERIC_READ表示可以对文件进行读操作。
  - GENERIC_WRITE表示可以对文件进行写操作。
- dwSharedMode：参数dwSharedMode表示可以被其他线程共享的访问权限。可选的参数主要有FILE_SHARE_READ和FILE_SHARE_WRITE。
- lpSecurityAttributes：同前面介绍的一些函数一样，参数lpSecurityAttributes在Windows CE中需要被设置成NULL。
- dwCreationDistribution：参数dwCreationDistribution的作用是通知CreateFile函数如何创建或者打开一个文件。其中可选的参数主要有：
  - CREATE_NEW：表示创建一个新的文件。如果在系统中该文件已经存在，则CreateFile函数调用失败。
  - CREATE_ALWAYS：表示不管什么情况，始终创建一个文件。
  - OPEN_EXISTING：表示打开一个已经存在的文件。
  - OPEN_ALWAYS：打开一个文件。当文件不存在的时候，则会创建相应的文件。
  - TRUNCATE_EXUSTING：表示打开一个文件，然后把这个文件缩短到0 bytes。
- dwFlagsAndAttributes：参数dwFlagsAndAttributes则定义了文件的属性。其可选值主要有：
  - FILE_ATTRIBUTE_NORMAL：函数默认的属性。这个属性可以被任意的其他属性所覆盖。
  - FILE_ATTRIBUTE_READONLY：设置文件为只读；当尝试往文件中写入数据的时候，将会发生失败。
  - FILE_ATTRIBUTE_ARCHIVE：标记文件的归档属性。
  - FILE_ATTRIBUTE_SYSTEM：文件为系统文件。也就是说，此文件对于整个Windows CE系统来说是非常重要的。
  - FILE_ATTRIBUTE_HIDDEN：隐藏文件或者目录。只有当把View All Files选项打开的时候，才可以看到这些被隐藏的文件。这一点和在桌面PC中使用的方式完全一样。
  - FILE_FLAG_WRITE_THROUGH：对文件的写操作不会被存储在Windows CE系统的内存中。
  - FILE_FLAG_RANDOM_ACCESS：针对随机访问对文件缓冲进行优化，也就是说告诉Windows CE系统，该文件将被随机访问而不是顺序访问。因此采用这个标志有助于系统确定用于该文件的高速缓存策略。

需要注意的是，在Windows CE中，参数dwFlagsAndAttributes不再支持FILE_ATTRIBUTE_OFFLINE、FILE_FLAG_OVERLAPPED等被Windows XP所支持的参数。这些标示主要包括：

- FILE_ATTRIBUTE_OFFLINE_SCAN
- FILE_FLAG_OVERLAPPED
- FILE_FLAG_SEQUENTIAL_SCAN
- FILE_FLAG_NO_BUFFRTING
- FILE_FLAG_DELETE_ON_CLOSE
- FILE_FLAG_POSIX_SEMANTICS
- FILE_FLAG_BACKUP_SEMANTICS

• hTemplateFile：参数hTemplateFile在Windows CE中不被系统所支持，因此被设置成0。

CreateFile函数被成功执行以后，将会返回已经打开文件的句柄。如果该函数调用失败，将会返回INVALID_HANDLE_VALUE。如果想获取此函数调用失败的原因，可以调用GetLastError函数。

函数CreateFile的使用实例为：

```
long hfile= CreateFile("c:\\WindowsCETest.txt",
                       GENERIC_READ,
                       FILE_SHARE_READ|FILE_SHARE_WRITE,
                       NULL,
                       OPEN_EXISTING,
                       FILE_FLAG_WRITE_THROUGH,
                       NULL)
```

2. 读/写文件

Windows CE支持标准的Win32 ReadFile和WriteFile函数。当函数成功执行以后，返回true，失败则返回false。

ReadFile函数的原型为：

```
BOOL ReadFile(HANDLE hFile,
              LPVOID lpBuffer,
              DWORD nNumberOfBytesToRead,
              LPDWORD lpNumberOfBytesToRead,
              LPOVERLAPPED lpOverlapped);
```

各个参数的含义为：

- hFile：参数hFile为指向预读文件的句柄。
- lpBuffer：参数lpBuffer为指向用来接收文件数据的缓冲区的指针。
- nNumberOfBytesToRead：参数nNumberOfBytesToRead为从文件中读取的字节数。
- lpNumberOfBytesToRead：参数lpNumberOfBytesToRead为指向这些字节数的指针。
- lpOverlapped：Windows CE中不支持lpOverlapped，因此在这里被设置成NULL。

需要注意的是，ReadFile函数不能读取超出文件大小的字节，如果将要读取的字节数已经超过了文件的大小，那么ReadFile函数仍旧会被成功执行，但是只会返回文件长度的字节数。

WriteFile函数的原型为：

```
BOOL WriteFile (HANDLE hFile,
                LPCVOID lpBuffer,
                DWORD nNumberOfBytedToWrite,
                LPDWORD lpNumberOfBytesWritten,
                LPOVERLAPPED lpOverlapped);
```

各个参数的含义为：

- hFile：参数hFile为指向预写文件的句柄。
- lpBuffer：参数lpBuffer为指向包含写入数据的缓冲区的指针。
- nNumberOfBytedToWrite：参数nNumberOfBytedToWrite为写入文件的字节数。
- lpNumberOfBytesWritten：参数lpNumberOfBytesWritten为指向这些字节数的指针。
- lpOverlapped：Windows CE中不支持lpOverlapped，因此在这里被设置成NULL。

3. 移动文件指针

如果想移动指向文件的指针，可以通过下面的函数来实现：

```
DWORD SetFilePointer (HANDLE hFile,
                      LONG lDistanceToMove,
                      PLONG lpDistanceToMoveHigh,
                      DWORD dwMoveMethod);
```

各个参数的含义为：

- hFile：参数hFile为指向文件的句柄。
- lDistanceToMove：低位指针的偏移。
- lpDistanceToMoveHigh：高位指针的偏移。
- dwMoveMethod：参数dwMoveMethod指出了指针从何处开始偏移。其可选值有：
  - FILE_BEGIN：从文件的开始位置。
  - FILE_CURRENT：从当前指针指向的位置开始。
  - FILE_END：从文件的末尾开始。

这个函数的返回值为指向新位置的指针。如果想查询当前指针的位置，则可以通过下面的方式来实现：

```
nCurrFilePtr=SetFilePointer (hFile,0,NULL,FILE_CURRENT);
```

代码的含义为从当前位置开始，移动文件指针0位，这样就得到了当前文件指针的位置。

4. 关闭文件

如果想在Windows CE中关闭一个文件，可以通过下面这个函数：

```
BOOL CloseHandle(HANDLE hObject);
```

当函数被成功执行以后，将会返回True，否则返回False。

5. 获取文件信息

在Windows CE.Net中可以通过GetFileAttributes函数，快捷、方便地获得文件的信息或者目录信息。

GetFileAttributes函数的原型为：

```
DWORD GetFileAttributes (LPCTSTR lpFileName);
```

这个函数的返回值可能为：

- FILE_ATTRIBUTE_COMPRESSED：表示文件是被压缩的。如果存储在ROM中的文件是采用压缩的方式存储的，就会占用相对较少的ROM空间。但是也就无法成文本地执行文件(execute-in-place)。在这里需要注意的是，Windows CE系统中基于RAM的对象存储文件总是处于压缩的状态，但是这些文件并没有被设置这个标志。
- FILE_ATTRIBUTE_ROMMODULE：表示存储在ROM中的文件处于本地执行状态，这些文件不能使用CreateFile函数打开。
- FILE_ATTRIBUTE_INROM：表示文件存储在ROM中。
- FILE_ATTRIBUTE_DIRECTORY：表示返回的是一个目录。
- FILE_ATTRIBUTE_TEMPORARY：表示此文件为临时存储文件。

可以通过下面这个函数来改变文件的属性：

```
BOOL SetFileAttributes (LPCTSTR lpFileName, DWORD dwFileAttribtues);
```

上面这两个函数都需要知道文件的句柄，而不是文件的名称，因此在使用它们之前需要通过CreateFile函数来获知文件的句柄。

6. 文件时间

标准的Win32函数支持三种文件时间，分别是文件被创建时的时间、最后一次访问文件的时间、最后一次修改文件的时间。要想获得这些文件时间，可以通过GetFileTime函数来实现。GetFileTime函数的原型为：

```
BOOL GetFileTime (HANDLE hFile,
                  LPFILETIME lpCreationTime,
                  LPFILETIME lpLastAccessTime,
                  LPFILETIME lpLastWriteTime);
```

各个参数的含义为：

- hFile：参数hFile为指向文件的句柄。
- lpCreationTime：参数lpCreationTime为指向FILETIME结构，以获取文件创建时间的指针，如果不想获得这个文件时间，只需要把这个参数设置成NULL即可。
- lpLastAccessTime：参数lpLastAccessTime为指向FILETIME结构，以获取文件最后一次访问时间的指针，这里指的访问包括文件的读、写等操作。如果不想获得这个文件时间，只需要把这个参数设置成NULL即可。
- lpLastWriteTime：参数lpLastWriteTime为指向FILETIME结构，以获取文件最后一次被写入时间的指针。如果不想获得这个文件时间，只需要把这个参数设置成NULL即可。

FILETIME结构体的函数原型为：

```
typedef struct _FILETIME { DWORD dwLowDateTime;
                           DWORD dwHighDateTime;
                         } FILETIME;
```

其中dwLowDateTime表示时间值的低32位字节；dwHighDateTime表示时间值的高32位字节。

可以通过FileTimeToSystemTime函数和FileTimeToLocalFileTime函数把GetFileTime函数或者其他函数返回的FILETIME结构转换成可以阅读的信息。

FileTimeToSystemTime函数的原型为：

```
BOOL FileTimeToSystemTime(const FILETIME *lpFileTIme,
                          LPSYSTEMTIME lpSystemTime);
```

FileTimeToLocaFileTime的函数原型为：

```
FileTimeToLocalFileTime(const FILETIME *lpFileTime,
                        LPFILETIME lpLocalFileTime);
```

当然也可以通过下面的函数来手工设置文件的时间：

```
BOOL SetFileTime (HANDLE hFile,
                  const FILETIME * lpCreateTime,
                  const FILETIME * lpLastAccessTime,
                  const FILETIME * lpLastWriteTime);
```

这个函数可以获取文件的句柄和三个FILETIME格式的时间。当只想设置其中的一种时间的时候，应该把其余几个时间参数设置成NULL。

7. 文件大小

可以通过下面的函数来获得文件的大小：

```
DWORD GetFileSize (HANDLE hFile, LPDWORD lpFileSizeHigh);
```

参数hFile为指向文件的句柄，如果不需要查找大于4GB的文件，只需把参数lpFileSizeHigh设置成NULL即可。

在Windows CE中，还有一个附加函数可以获得更多丰富的文件信息。这个函数为GetFileInformationByHandle。其函数原型为：

```
GetFileInformationByHandle (HANDLE hFile,
                            LPBY_HANDLE_FILE_INFORMATION lpFileInformation);
```

这个函数将会获取已经打开的文件的句柄和一个指向BY_HANDLE_FILE_INFORMATION结构的指针。如果这个函数被调用成功，将会返回TRUE。

BY_HANDLE_FILE_INFORMATION结构定义为：

```
typedef struct _BY_HANDLE_FILE_INFORMATION{
    DWORD       dwFileAttributes;
    FILETIME    ftCreationTime;
    FILTTIME    ftLastAccessTime;
    FILETIME    ftLastWriteTime;
    DWORD       dwVolumeSerialNumber;
    DWORD       nFileSizeHigh;
    DWORD       nFileSizeLow;
    DWORD       nNumberOfLinks;
    DWORD       nFileIndexHigh;
    DWORD       nFileIndexLow;
    DWORD       dwOID;
} BY_HANDLE_FILE_INFORMATION
```

从上面列出的名字中可以很清楚地看到一些需要的数据，如文件的创建时间、最后一次访问时间等。

7.1.4 内存映射文件

前面已经介绍了Windows CE中的虚拟内存，同虚拟内存一样，Windows CE中的内存映射文件也能够保留一个虚拟地址空间，并将物理存储器提交到该区域。在程序设计中，任何应用程序都需要对文件进行操作。那么应用程序究竟应该是打开文件、关闭文件还是读取文件呢？还是首先打开文件，然后使用一种缓冲算法，从文件的各个不同部分进行读写或者写入呢？正是为了解决上述问题，微软引入了内存映射文件的概念。

内存映射文件提供了一种完全不同的对文件进行读取和写入的方法，对于标准的文件来说，文件是以数据流的形式被读取的。当要访问文件不同部分中的字节的时候，文件的指针必须首先移动到文件中第一个字节的位置上，然后读取该数据。读取完成以后，文件指针再移动到其他的字节处，接着进行文件的读取。

1. 概述

在早期的内存映射文件技术中，它并不是为了提交物理内存供调用者使用，而是为了提交永久存储器上的文件。由于内存映射文件减少了程序加载所需的时间，因此非常适合用来加载可执行文件和动态链接库文件。

内存映射文件一般被用于下面三种不同的目的：

1）可以使用内存映射文件来访问磁盘上的数据文件，这样就不必对文件进行I/O操作，因此就不必对文件内容进行缓存。

2）系统通过内存映射文件，可以方便地加载和执行可执行文件（.exe）和动态链接库（.dll）文件。这样就可以大大地节省应用程序启动时所需的时间和页文件空间。

3）可以通过内存映射文件，使在同一台计算机上运行的多个进程可以共享数据资源。内存映射文件已经成为了一台计算机上多个进程之间相互通信的最有效的方法。

在新版本的Windows CE中，系统支持无名的和有名的内存映射文件。建议在软件的开发过程中，如果需要进行大容量的文件读写操作，申请开通大容量的虚拟地址空间，或者需要在几个不同的进程之间通信，最好采用内存映射文件技术。因为合理地运用内存映射文件技术会大大地提高工作效率。

对于系统中的内存映射文件，由于其已经被映射到了内存中的某一个区域，因此不再需要调用FileRead和FileView函数，只需要对被映射到的内存区域进行读取和写入即可。此时，对内存的更新将会自动地反映到文件本身，因此也就使得其对文件的读取和写入变得非常的容易。

2. 相关函数

在Windows Embedded CE 6.0及其后续的版本中，如果想要打开一个供内存映射技术访问的文件，首先需要调用CreateFile函数来打开或者创建一个供内存映射技术访问的文件。而在Windows CE 5.0或者更早的版本中，这一步骤一般情况下是用CreateFileForMapping函数来实现的。同桌面操作系统类似，后续版本的Windows CE做到了和前面各个版本的兼容，因此在Windows Embedded CE 6.0中，系统也提供了对CreateFileForMapping函数的支持，只不过CreateFile函数使用得会更多一些。

CreateFileForMapping函数的函数原型为：

```
HANDLE CreateFileForMapping (LPCTSTR lpFileName,
                             DWORD dwDesiredAccess,
                             DWORD dwShareMode,
                             LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                             DWORD dwCreationDisposition,
                             DWORD dwFlagsAndAttributes,
                             HANDLE hTemplateFile);
```

当这个函数调用成功以后，将返回文件的句柄。

CreateFileForMapping函数中各个参数的含义为：

- lpFileName：参数lpFileName为指定文件的名称或者文件的路径。在这里需要注意的是，Windows CE中的文件路径格式是没有盘符符号的。
- dwDesiredAccess：参数dwDesiredAccess指定了文件的访问方式，是读方式还是写方式。
- dwShareMode：参数dwShareMode指定了文件的共享方式。

- lpSecurityAtttributes：同前面介绍的那些函数一样，在Windows CE中这个参数被设置成NULL。
- dwCreationDisposition：参数dwCreationDisposition表示此函数的功能是创建文件还是打开文件。
- dwFlagsAndAttributes：参数dwFlagsAndAttributes指定了文件的属性。
- hTemplateFile：同前面介绍的函数一样，在Windows CE中此参数将被忽略。

在创建或者打开一个内存映射文件以后，接下来就需要为指定的内存映射文件创建一个文件映射对象。可以通过CreateFileMapping函数来实现，其函数原型为：

```
HANDLE CreateFileMapping (HANDLE hFile,
                          LPSECIRITY_ARRIBUTED lpFileMappingAttributes,
                          DOWRD flProtect,
                          DWORD dwMaximumSizeHigh,
                          DWORD dwMaximumSizeLow,
                          LPCTSTR lpName);
```

这个函数可以创建一个有名的或者无名的内存映射文件对象，当函数被成功执行以后，将会返回创建的内存映射文件对象的句柄。其各个参数的含义为：

- hFile：参数hFile为指定文件的句柄。
- lpFileMappingAttributes：参数lpFileMappingAttributes为安全属性，在Windows CE中需要被设置成NULL。
- flProtect：参数flProtect指定了要映射文件的保护属性，是只读的还是可读或者可写的。
- dwMaximumSizeHigh：参数dwMaximumSizeHigh指定了内存映射文件的高32位。
- dwMaximumSizeLow：参数dwMaximumSizeLow指定了内存映射文件的低32位。
- lpName：参数lpName指定了内存映射文件对象的名称。

在创建了内存映射文件对象以后，接下来就需要将文件对象映射到地址空间，可以通过函数MapViewOfFile来实现。其函数原型为：

```
LPVOID MapViewOfFile (HANDLE hFileMappingObject,
                      DWORD dwDesiredAccess,
                      DWORD dwFileOffsetHigh,
                      DWORD dwFileOffsetLow,
                      DWORD dwNumberOfBytesToMap);
```

这个函数可以保留一段足够的地址空间，并将永久存储器上的文件数据映射到这个地址空间。映射后这段地址空间叫做文件视图，映射范围可以覆盖部分文件，也可以覆盖全部文件。各个参数的含义为：

- hFileMappingObject：参数hFileMappingObject指定了内存映射文件对象的句柄。
- dwDesiredAccess：参数dwDesiredAccess限定了访问权限。
- dwFileOffsetHigh：参数dwFileOffsetHigh指定了映射区域开始的高32位位置。
- dwFileOffsetLow：参数dwFileOffsetLow指定了映射区域开始的低32位位置。
- dwNumberOfBytesToMap：参数dwNumberOfBytesToMap指定了映射区域的大小。

在这里需要注意的是，dwFileOffsetHigh参数和dwFileOffsetLow 参数指定的64位开始访问地

址在Windows CE系统中可以不是64KB的整数倍。而在其他的Windows操作系统中，则必须是以64KB为单位。

操作完成以后，撤销文件映射视图的函数为：

```
BOOL UnmapViewOfFile (LPCVOID lpBaseAddress);
```

里面唯一的参数表示视图的首地址。

下面的一段代码展示了Windows Embedded CE 6.0中实现映射文件操作的过程：

```
HANDLE hfile, hFileMap;
PBYTE pFileMem;
TCHAR szFileName[MAX_PATH];

hFile=CreateFile(szFileName, GENERIC_WRITE, FILE_SHARR_TEAD, NULL,
                 OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL |
                 FILE_FLAG_RANDOM_ACCESS, 0);


If(hFile != INVALID_HANDLE_VALUE) {
    pFileMap=CreateFileMapping(hFile, NULL, PACE_READWRITE, 0, 0, 0);
    if(hFileMap){
        pFileMem=(PBYTE)MapViesOfFile(hFileMap, FILE_MAP_WROTE.
                                      0, 0, 0);

        if(pFileMem){
        //
        //对文件数据进行相应的操作
        //
        UnmapViewOfFile(pFileMem);
        }
        CloseHandle (hFileMap) ;
    }
    CloseHandle(hFile);
}
```

学习完上面的代码以后，回过头来再看一下Windows Embedded CE 6.0中实现映射文件的操作步骤。

1）通过CreateFile函数打开或者创建一个供内存映射技术访问的文件。

2）通过CreateFileMapping函数为指定的内存映射文件创建一个文件映射对象。

3）通过MapViewOfFile函数将文件对象映射到地址空间。

4）进行相应的读写操作。

5）通过UnmapViewofFile函数结束整个操作。

7.1.5 文件管理

前面已经介绍了如何读取和改写一个文件，在接下来的内容中，将介绍如何对文件进行复制、移动等操作。

1. 文件的复制

如果想复制一个文件，可以通过下面这个函数来实现：

```
BOOL CopyFile (LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName,
               BOOL bFailIfExists);
```

各个参数的含义为：

- lpExistingFileName：参数lpExistingFileName为源文件的文件名称。
- lpNewFileName：参数lpNewFileName为目标文件的文件名称。
- bFailIfExists：当参数bFailIfExists为True，lpNewFileName的名称在系统中已经存在的时候，CopyFile函数调用失败；当参数bFailIfExists为false，lpNewFileName的名称在系统中已经存在的时候，CopyFile函数将会用源文件覆盖已经存在的同名文件，此时CopyFile函数调用成功。

2. 文件的移动

如果想在Windows CE系统中移动一个文件，则可以通过下面这个函数来实现：

```
BOOL MoveFile(LPCTSTR lpExistingFileName,LPCTSTR lpNewFileName);
```

各个参数的含义为：

- lpExistingFileName：参数lpExistingFileName为源文件的名称。
- lpNewFileName：参数lpNewFileName为目的文件的名称或者路径。在这里需要注意的是，该参数指明的文件必须是系统中不存在的。

文件的移动可以在对象存储的内部进行，也可以在对象存储和外部存储之间进行。同时，如果把文件名称设置成不同，而保持相同的路径，就实现了文件的重命名。

3. 文件的删除

在Windows Embedded CE 6.0中，要想删除一个文件可以通过下面这个函数：

```
BOOL DeleteFile(LPCTSTR lpFileName);
```

里面的参数为需要删除的文件的名称。当将要被删除的文件处于被打开的状态的时候，此函数将调用失败。

4. 创建临时文件

在Windows Embedded CE 6.0程序设计中经常需要创建临时文件，可以通过下面这个函数来实现这个功能：

```
UNIT GetTempFileName (LPCTSTR lpPathName, LPCTSTR lpPrefixString,
                      UNIT uUnique, LPCTSTR lpTempFileName);
```

各个参数的含义为：

- lpPathName：参数lpPathName为临时文件的路径，它可以是当前路径或者已经存在的一个路径。
- lpPrefixString：参数lpPrefixString为文件名称的前缀。这个字符串的前3个字符为临时文件名称的前缀。
- uUnique：当参数uUnique为0的时候，函数将创建一个唯一名称的临时文件；不为0的时候，将不受这个限制。

• lpTempFileName：参数lpTempFileName为临时文件的名称。

5. 文件查找

如果想在Windows Embedded CE 6.0系统中查找一个文件，可以通过下面这个函数：

```
HANDLE FindFirstFile (LPCTSTR lpFileName, LPWIN32_FIND_DATA lpFindFileData );
```

其中参数lpFileName为查找文件的名称，参数lpFindFileData为一个指向LPWIN32_FIND_DATA结构的指针。当FindFirstFile函数没有找到想要的文件或者路径的时候，函数将返回INVALID_HANDLE_VALUE。

如果想要查找下一个文件，则可以通过下面的函数：

```
BOOL FindNextFile (HANDLE hFindFile, LPWIN32_FIND_DATA lpFindFileData);
```

参数hFindFile为FindFirstFile函数返回值的句柄，参数lpFindFileData为一个指向LPWIN32_FIND_DATA结构的指针。

当想结束文件查找的时候，就可以通过下面的函数来实现：

```
BOOL FindClose (HANDLE hFindFile);
```

这个函数将会接收FindFirstFile函数返回的句柄。

上面提到的LPWIN32_FIND_DATA结构的原型为：

```
typedef struct _WIN32_FIND_DATA {DWORD dwFileAttributes;
                                 FILETIME ftCreationTime;
                                 FILETIME ftLastAccessTime;
                                 FILETIME ftLastWriteTime;
                                 DWORD nFileSizeHigh;
                                 DWORD nFileSizeLow;
                                 DWORD dwOID;
                                 TCHAR cFileName[MAX_PATH];
                                }WIN32_FIND_DATA;
```

其中参数dwFileAttributes为找到文件的属性，它可以是下列值中的任一个或几个：

• FILE_ATTRIBUTE_ARCHIVE：表示找到的文件是档案文件（用来备份数据的文件）。
• FILE_ATTRIBUTE_COMPRESSED：表示文件或者目录是被压缩的。
• FILE_ATTRIBUTE_DIRECTORY：表示返回的文件句柄是一个路径。
• FILE_ATTRIBUTE_HIDDEN：表示文件名称或者路径是隐藏的。
• FILE_ATTRIBUTE_ENCRYPTED：表示文件或者路径是加密的。
• FILE_ATTRIBUTE_INROM：表示找到的文件存储在ROM中。
• FILE_ATTRIBUTE_READONLY：表示找到的文件或者路径是只读的。
• FILE_ATTRIBUTE_NORMAL：表示找到的文件或者路径没有别的属性设置。
• FILE_ATTRIBUTE_REPARSE_POINT：表示找到的文件是双向关联的。
• FILE_ATTRIBUTE_TEMPORARY：表示找到的是一个临时存储文件。

再来看看其他几个参数所代表的含义：

• ftCreationTime：参数ftCreationTime表示文件被创建的时间。
• ftLastAccessTime：参数ftLastAccessTime表示文件最后被访问的时间。
• ftLastWriteTime：参数ftLastWriteTime表示文件最后被写入的时间。

- nFileSizeHigh：参数nFileSizeHigh表示文件大小的高32位。
- nFileSizeLow：参数nFileSizeLow表示文件大小的低32位。
- dwOID：参数dwOID指定了文件的标示对象。
- cFileName：参数cFileName表示文件的名称。

7.1.6 存储管理器

在Windows Embedded CE 6.0中，存储管理器（Storage Manager）是通过Filesys.dll来实现的，具体的功能可以通过调用这个Filesys.dll来完成。例如，当CF或者SD卡被加载到Windows Embedded CE 6.0设备的时候，就需要使用到存储管理器。存储管理器由以下几个部分组成：

1）块驱动程序：它们是物理存储介质的设备驱动程序，提供了对数据存储的随机寻址块的访问。

2）分区驱动程序：为单个存储设备上的多个分区提供管理。Windows CE系统允许物理磁盘包含多个分区，并且每个分区可以格式化为不同的文件系统。分区驱动程序实际上是存储驱动程序的转换器。它公开与存储驱动程序相同的接口，并将分区的块地址转换为存储设备块的真实地址。然后，它将调用传递给存储驱动程序。

3）文件系统驱动程序：这些驱动程序将存储设备上的数据组织为文件和文件夹。

4）文件过滤器：文件过滤器用于处理对文件系统的调用，此后，文件系统才能获得这些调用。这就允许对文件访问进行某些特殊的处理，以便进行数据加密、压缩和使用统计数据进行监视。

图7-1展示了Windows Embedded CE 6.0中文件系统各个组件的关系。图7-2展示了存储管理器和它的相关组件。

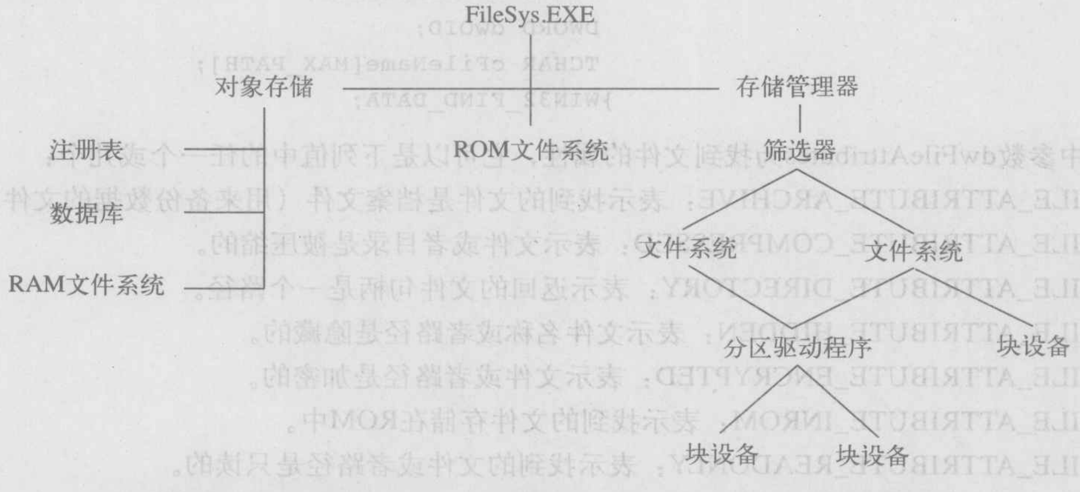


图7-1 Windows Embedded CE 6.0中文件系统各个组件的关系

需要注意的是，并非所有文件系统驱动程序都使用了物理设备，即使使用，也可能没有使用分区驱动程序。这就提供了巨大的灵活性。例如，负责提供网络共享访问的网络重定向器使用WinSock 通过网络与远程服务器通信，并且它在 Windows CE设备上没有物理磁盘。

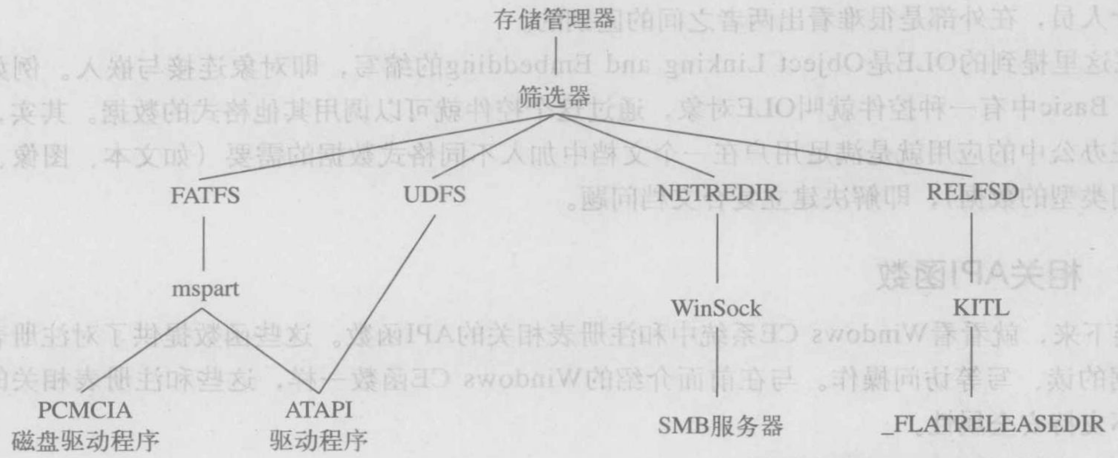


图7-2 存储管理器和它的相关组件

7.2 注册表

在介绍完Windows Embedded CE 6.0中的文件系统以后，再来看一下另一个非常重要的内容——注册表。

7.2.1 概述

注册表实际上可以被看作是一个存储系统资源的数据库，里面通常包含了应用程序和Windows系统本身的一些配置信息。尽管Windows CE中的注册表和其他的Windows系统有一点类似，但是在一些函数及其使用上仍旧有很大的不同。在所有的Windows系统中，注册表都是由若干个键及其键值所组成。这一点类似文件系统，键相当于目录，而值则相当于文件。

在Windows CE中，整个系统的注册表形成了一个有三个节点的森林，每个根键下都存储一类配置信息。三个根键为：

- HKEY_LOCAL_MACHINE
- HKEY_CLASSES_ROOT
- HKEY_CURRENT_USER

其中HKEY_LOCAL_MACHINE是用来存储硬件和驱动程序的配置信息，HKEY_CLASSES_ROOT用来存储文件类型和OLE配置信息，而HKEY_CURRENT_USER则用来存储用户配置信息。例如，Internet Explorer浏览器的配置信息就存放在下面这个目录：

HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer

图7-3展示了Windows CE中的注册表。

Windows CE支持两种类型的注册表，一种是基于对象存储的，一种是与Windows XP相同的注册表数据库结构。在基于对象存储的注册表结构中，注册表的所有数据使用对象存储方式存储，这样方式比较适合电池、内存等硬件系统。而另外一种注册表则适合于当系统具有多用户支持的时候。作为程

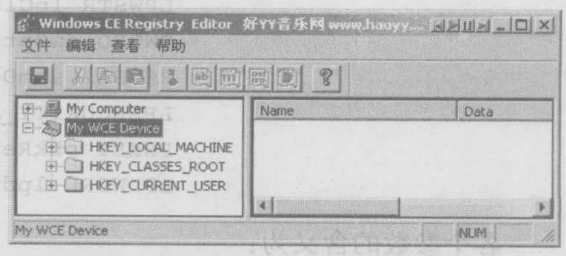


图7-3 Windows CE中的注册表

序设计人员，在外部是很难看出两者之间的区别的。

在这里提到的OLE是Object Linking and Embedding的缩写，即对象连接与嵌入。例如，在Visual Basic中有一种控件就叫OLE对象，通过这个控件就可以调用其他格式的数据。其实，OLE技术在办公中的应用就是满足用户在一个文档中加入不同格式数据的需要（如文本、图像、声音等不同类型的数据），即解决建立复合文档问题。

7.2.2 相关API函数

接下来，就看看Windows CE系统中和注册表相关的API函数。这些函数提供了对注册表中存储数据的读、写等访问操作。与在前面介绍的Windows CE函数一样，这些和注册表相关的函数同样不支持安全属性。

1. 打开或者创建注册表中的键

可以通过下面的函数来打开一个注册表中的键：

```
LONG RegOpenKetEx (HKEY hKey,LPCWSTR lpszSubKey,DWORD ulOptions,
                   REGSAM SamDesired, PHKEY phResult);
```

各个参数的含义为：

- hKey：参数hKey为打开键的句柄。这个句柄由RegCreateKeyEx函数或者 RegOpenKeyEx函数返回。或者为下面列表中显示出来的预定义键：
  - HKEY_CLASSES_ROOT
  - HKEY_CURRENT_CONFIG
  - HKEY_CURRENT_USER
  - HKEY_LOCAL_MACHINE
  - HKEY_USERS
- lpszSubKey：参数lpszSubKey为指向被打开键名称的指针。它应该指向参数hKey指向键的子键。
- ulOptions：参数ulOptions在Windows CE中将被设置成0。
- SamDesired：参数SamDesired在Windows CE中将被设置成NULL。
- phResult：参数phResult为指向特定变量的指针，这个特定变量主要用来存储打开键的句柄。

在Windows CE中，也可以通过另外一个函数来打开注册表的键：

```
LONG RegCreateKeyEx (HKEY hKey,
                     LPCWSTR lpSubKey,
                     DWORD Reserved,
                     LPWSTR lpClass,
                     DWORD dwOptions,
                     REGSAM samDesired,
                     LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                     PHKEY phkResult,
                     LPDWORD lpdwDisposition );
```

各个参数的含义为：

- hKey：参数hKey为指向当前打开键的句柄，或者为下列预定义的句柄值：

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS

• lpSubKey：参数lpSubKey为指向子键名称的指针。

• Reserved：参数Reserved在Windows CE中被设置成0。

• lpClass：参数lpClass为指向键类名称的指针，当不需要指定类名的时候，这个参数被设置成NULL。

• dwOptions：和前面那个函数一样，参数dwOptions在Windows CE中需要被设置成0。

• samDesired：参数samDesired需要被设置成NULL。

• lpSecurityAttributes：参数lpSecurityAttributes在Windows CE中被设置成NULL。

• phkResult：参数phkResult为指向特定变量的指针，这个特定变量主要用来存储打开键的句柄。

• lpdwDisposition：参数lpdwDisposition为指向特性变量的指针。这个变量可以用来存储下列几个值：

- REG_CREATED_NEW_KEY：表示创建一个新的键。
- REG_OPENED_EXISTING_KEY：表示打开一个已经存在的键。

RegOpenKeyEx函数和RegCreateKeyEx函数的区别在于当需要打开的键在注册表中不存在的时候，RegCreateKeyEx函数可以新创建这个键，而RegOpenKeyEx函数则不能。

2. 读取注册表的值

在打开一个注册表键以后，就可以通过下面的函数读取其中的键值：

```
LONG RegQueryValueEx (HKEY hKey,
                      LPCWSTR lpValueName,
                      LPDWORD lpReserved,
                      LPDWORD lpType,
                      LPBYTE lpData,
                      LPDWORD lpcbData );
```

各个参数的含义为：

• hKey：参数hKey为打开键的句柄，或者为下列预定义的句柄：

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS

• lpValueName：参数lpValueName表示需要查询的键值的名称。

• lpReserved：参数lpReserved在Windows CE中被设置成NULL。

• lpType：参数lpType为指向变量的数据类型的指针，其可选值大致包括：

- REG_BINARY：表示数据类型为二进制数据。
- REG_DWORD：表示是一个32位的数字。
- REG_DWORD_LITTLE_ENDIAN：表示从低位开始的32位数值。
- REG_DWORD_BIG_ENDIAN：表示从高位开始的32位数值。

  - REG_NONE：表示未定义的数据类型。
  - REG_RESOURCE_LIST：表示设备驱动资源列表。
- lpData：参数lpData为指向数据缓冲区的指针。
- lpcbData：参数lpcbData表示指向数据大小的指针。

3. 向注册表中写入数据

可以通过下面的函数向注册表中写入数值：

```
LONG RegSetValueEx (HKEY hKey,
                    LPCWSTR lpValueName,
                    DWORD Reserved,
                    DWORD dwType,
                    const BYTE *lpData,
                    DWORD cbData );
```

各个参数的含义为：

- hKey：参数hKey为指向打开注册表键的句柄，或者为下列预定义键：
  - HKEY_CLASSES_ROOT
  - HKEY_CURRENT_USER
  - HKEY_LOCAL_MACHINE
  - HKEY_USERS
- lpValueName：参数lpValueName为需要写入数据的键的名称。
- Reserved：参数Reserved在Windows CE中需要被设置成0。
- dwType：参数dwType为需要存储的数据类型。其取值可参照上面的RegQueryValueEx函数介绍。
- *lpData：参数*lpData为指向存储数据的缓冲区的指针。
- cbData：参数cbData为数据的大小。

4. 删除注册表键和键值

如果想在Windows Embedded CE 6.0中删除注册表键，可以通过下面的函数：

```
LONG RegDeleteValue (HKEY hKey, LPCWSTR lpValueName);
```

其中参数hKey的含义和前面几个函数一样，参数lpValueName表示需要被删除值。

如果想在程序设计的过程中收集比较全面的注册表键信息，可以通过下面这个函数：

```
LONG RegQueryInfoKey (HKEY hKey, LPWSTR lpClass, LPDWORD lpcbClass,
                      LPDWORD lpReserved,  LPDWORD lpcSubKeys,
                      LPDWORD lpcbMaxSubKeyLen,
                      LPDWORD lpcbMaxClassLen,
                      LPDWORD lpcValues,
                      LPDWORD lpcbMaxValueNameLen,
                      LPDWORD lpcbMaxValueLen,
                      LPDWORD lpcbSecurityDescriptor,
                      PFILETIME lpftLastWriteTime );
```

各个参数的含义为：

- hKey：参数hKey为注册表键的句柄。

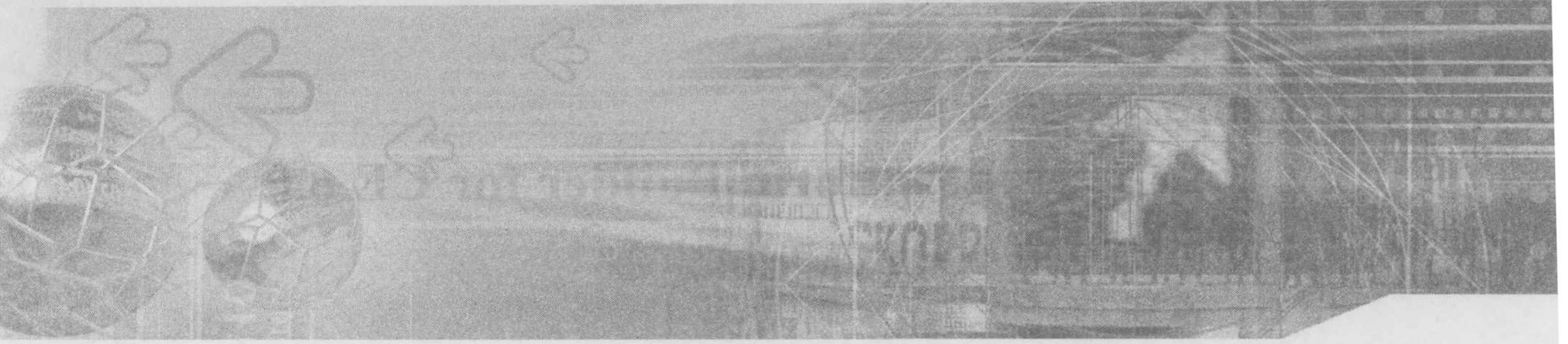
- lpClass：参数lpClass为指向存储注册表键类的缓冲区的指针。
- lpcbClass：参数lpcbClass为指向存储缓冲区大小的变量的指针。
- lpReserved：参数lpReserved在Windows CE中需要被设置成NULL。
- lpcSubKeys：参数lpcSubKeys为指向子键的指针。
- lpcbMaxSubKeyLen：参数lpcbMaxSubKeyLen为指向存储子键键名最大长度的变量的指针。
- lpcbMaxClassLen：参数lpcbMaxClassLen指向存储子键长度最大值的变量的指针。
- lpcValues：参数lpcValues为和注册表键相关的数值的个数。
- lpcbMaxValueNameLen：参数lpcbMaxValueNameLen为注册表键值名称的最大值。
- lpcbMaxValueLen：参数lpcbMaxValueLen为注册表键值长度的最大值。
- 参数lpcbSecurityDescriptor和lpftLastWriteTime在Windows CE中需要被设置成NULL。

5. 关闭注册表键

如果想关闭注册表键可以通过下面的函数：

```
LONG RegCloseKey (HKEY hKey);
```

函数中唯一的参数为注册表键的句柄。



第二部分

Windows Embedded CE 6.0 程序设计实例

第8章　使用Platform Builder for CE 6.0进行内核定制

8.1　Platform Builder for CE 6.0简介

本章将学习如何使用Platform Builder for CE 6.0进行Windows Embedded CE 6.0系统的内核定制。在学习之前，先看一下什么是Platform Builder。

8.1.1　Platform Builder for CE 6.0的主要功能

Platform Builder是由微软公司发布的一款用于定制Windows CE操作系统的集成开发环境。开发人员可以使用这个集成开发环境完成嵌入式操作系统的定制、编译和调试，还可以在这个集成开发环境中进行应用软件、驱动程序的开发等。

Platform Builder主要包括下面一些开发工具：

1）创建平台向导：它的主要作用是创建一个新的嵌入式实验平台。

2）创建BSP向导：它的主要作用是创建一个新的BSP。

3）导出SDK向导：它的主要作用是导出一个定制的软件开发包。

4）自动的依赖性检测：它的主要作用是保证支持一个平台所需的特性，都可以被完整地包含到操作系统中。

5）目录：在Platform Builder的目录中，列出了可以用来定制一个系统平台的一些特性。

6）导出目录向导：使用这个工具可以向其他的Platform Builder用户导出一个定制的目录。

7）基本的配置：它是创建一个新操作系统的起点。

8）Windows CE Test Kit：它的主要作用是Windows CE中驱动程序的测试。

9）内核调试器：它的作用是调试一个定制的操作系统的镜像。

10）应用程序调试器：它的作用是在定制的操作系统镜像上调试相应的应用程序。

11）模拟器工具：使用这个工具可以让程序设计人员在开发工作区上测试嵌入式平台或者编写的应用程序。类似使用Visual Studio中的智能设备模拟器。

12）远程调试工具：它的主要作用是在一个真实的目标设备上进行相应程序的调试。

在Platform Builder的早期版本中，它是作为一个独立的软件而单独存在的。伴随着Visual Studio 2005的发布，以及微软公司对嵌入式领域的研发投入日益加大，在新的版本中，Platform Builder则变成了Visual Studio 2005的一个插件。

图8-1展示了Platform Builder 4.2的用户界面，图8-2展示了Platform Builder for CE 6.0的图形用户界面。

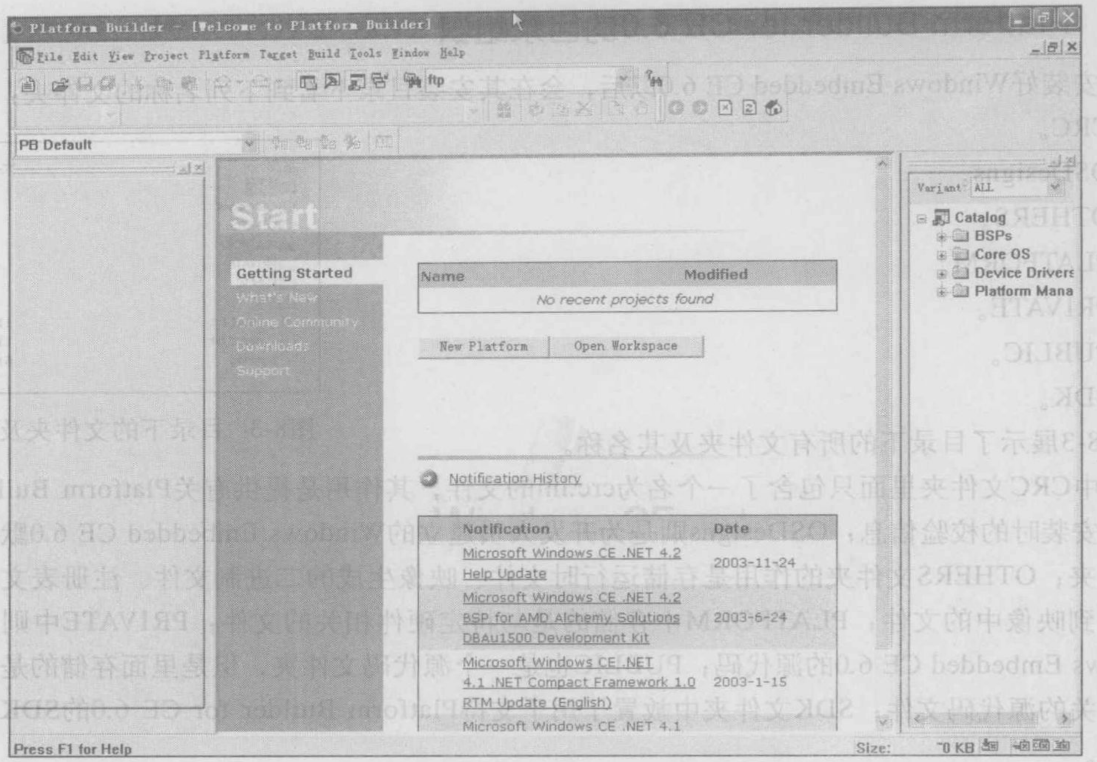


图8-1　Platform Builder 4.2的用户界面

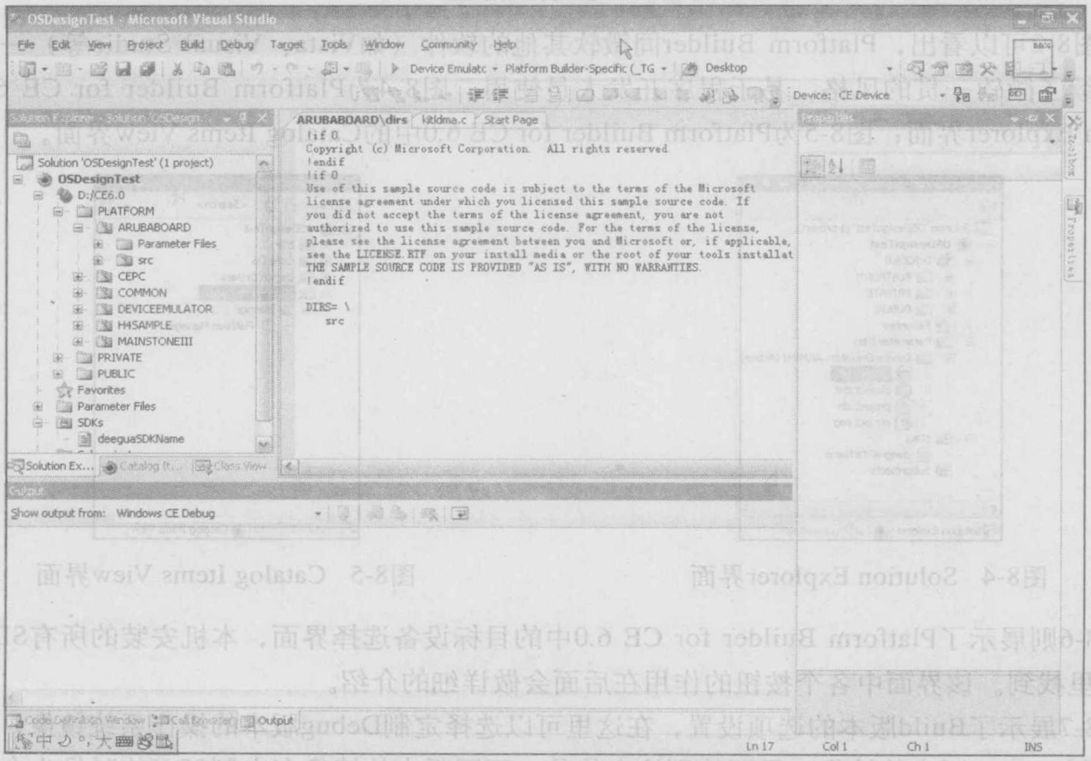


图8-2　Platform Builder for CE 6.0界面

8.1.2 Platform Builder for CE 6.0的目录组织

当安装好Windows Embedded CE 6.0以后，会在其安装目录下看到下列名称的文件夹：

- CRC。
- OSDesigns。
- OTHERS。
- PLATFORM。
- PRIVATE。
- PUBLIC。
- SDK。

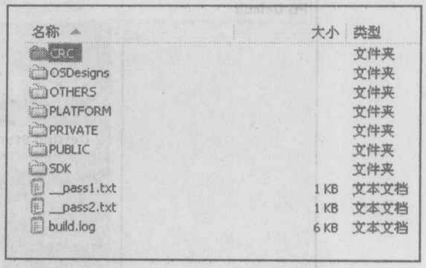


图8-3 目录下的文件夹及其名称

图8-3展示了目录下的所有文件夹及其名称。

其中CRC文件夹里面只包含了一个名为crc.ini的文件，其作用是提供有关Platform Builder for CE 6.0安装时的校验信息；OSDesigns则是为开发人员建立的Windows Embedded CE 6.0默认存储的文件夹；OTHERS文件夹的作用是存储运行时文件、映像生成的二进制文件、注册表文件等需要加入到映像中的文件；PLATFORM中存储的是与特定硬件相关的文件；PRIVATE中则包含了Windows Embedded CE 6.0的源代码；PUBLIC也是一个源代码文件夹，但是里面存储的是与硬件平台无关的源代码文件；SDK文件夹中放置了用于支持Platform Builder for CE 6.0的SDK工具和库文件。

8.1.3 Platform Builder for CE 6.0的用户界面

从图8-2可以看出，Platform Builder同微软其他的软件（如Vista、Visual Studio等）一样，其界面延续了微软一贯的风格，易于程序开发人员使用。图8-4为Platform Builder for CE 6.0中的Solution Explorer界面；图8-5为Platform Builder for CE 6.0中的Catalog Items View界面。

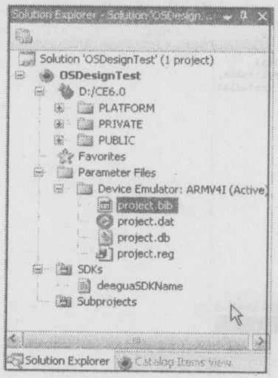


图8-4 Solution Explorer界面

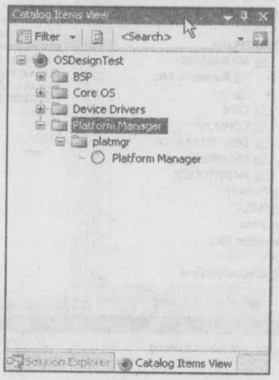


图8-5 Catalog Items View界面

图8-6则展示了Platform Builder for CE 6.0中的目标设备选择界面，本机安装的所有SDK都可以在这里找到。该界面中各个按钮的作用在后面会做详细的介绍。

图8-7展示了Build版本的选项设置，在这里可以选择定制Debug版本的操作系统镜像，也可以选择定制Release版本的镜像。需要特别注意的是，不同版本的镜像在定制SDK的时候也会导致定制的选项的不同，具体的应用在本章后面会有详细的介绍。

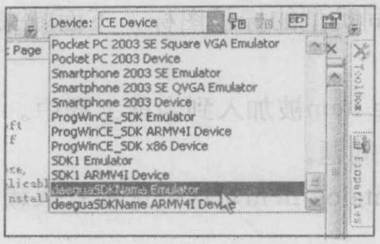


图8-6 Device选择界面

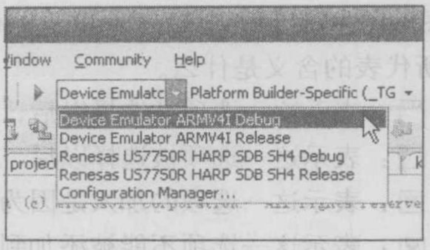


图8-7 Build版本选项设置

同微软发布的其余开发工具类似，在Platform Builder for CE 6.0下方也集成了信息显示对话框，在这里可以看到相关的编译信息等内容。图8-8展示了这一信息显示界面。

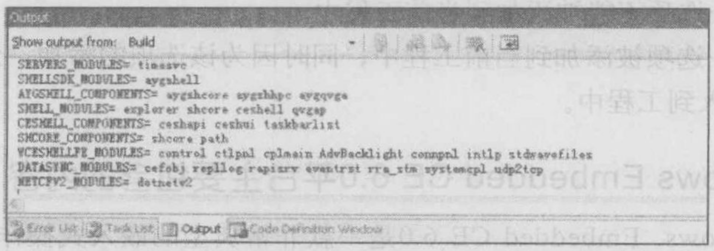


图8-8 Platform Builder for CE 6.0中的调试信息显示界面

8.1.4 Catalog Items View

Windows Embedded CE 6.0中包含的所有特性都是一些组件在逻辑上的有序组合。因此利用Catalog Items View窗口可以向Windows Embedded CE 6.0的内核添加或者删除一些特性。图8-9展示了Catalog Items View窗口。

在Catalog Items View窗口中有一个过滤器（Filter），通过这个过滤器可以改变Catalog Items View中的显示内容。这个过滤器中的三个选项分别为：

- User-selected Catalog Items Only：只显示用户选择的选项信息。
- User-selected Catalog Items and Dependencies：显示用户选择的选项信息及其相关的依赖选项信息。
- All Catalog Items in Catalog：所有的Catalog信息。

图8-10展示了Catalog Items View中的过滤器（Filter）选项。

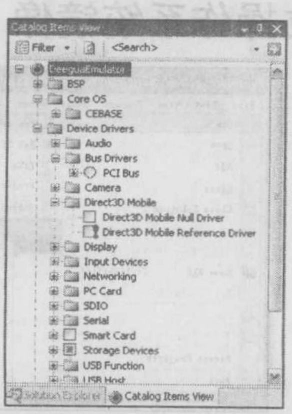


图8-9 Catalog Items View窗口

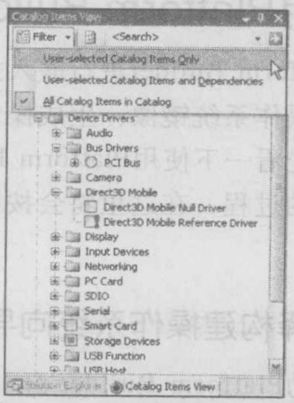


图8-10 Catalog Items View中的过滤器

从图8-10中可以看到，在Catalog Items View中存在不同类型的标示图标，下面就来看一下各个图标所代表的含义是什么。

1）：表示这一选项被选择以后，将会有多个Catalog Item被加入到当前工程中。

2）：表示这一选项被选择以后，可能会引起其他的问题。

3）：表示这一选项的加入是因为其他已经添加的Catalog Item对这个选项有依赖。

4）：表示这一选项不能被添加到当前工程中。

5）：表示这一选项被添加到当前工程中，同时因为该选项的添加，会有其他Catalog Item由于依赖关系而被加入到工程中。

6）：表示这一选项被选择以后，将有一个Catalog Item被加入到当前工程中。

7）：表示这一选项不能被添加到当前工程中。

8）：表示这一选项被添加到当前工程中，同时因为该选项的添加，会有其他Catalog Item由于依赖关系而被加入到工程中。

8.1.5 定制Windows Embedded CE 6.0平台主要包含的内容

我们知道，Windows Embedded CE 6.0是一款非常典型的嵌入式操作系统。那么，当使用Platform Builder for CE 6.0进行嵌入式系统平台定制的时候，一般都包含哪些内容呢？

定制的平台主要包括：

1）操作系统的镜像：主要包括Platform Builder中包含的现成的可选项。

2）生成的BSP：主要包括驱动程序和其他一些Platform Builder中包含的现成的可选项。

3）新添加的组件：主要包括一些应用程序。

4）修改平台的组件。

5）设置的平台属性和环境变量。

6）设置的远程连接属性，使我们可以方便地下载操作系统。

7）下载操作系统：主要包括把生成的操作系统下载到硬件平台和把操作系统下载到系统自带或自定义的模拟器。

8）对生成的操作系统进行相应的调试：主要包括在模拟器上进行调试和在硬件平台上进行调试。

8.2 使用Platform Builder for CE 6.0建立操作系统镜像

使用Platform Builder进行嵌入式操作系统定制的过程主要包括建立操作系统镜像、生成操作系统镜像和调试操作系统镜像。首先看一下使用Platform Builder for CE 6.0建立操作系统镜像的过程，在这里将会按照工程向导的提示步骤一一介绍。

8.2.1 选择构建操作系统向导

首先启动Platform Builder for CE 6.0，然后选择File→New→Project，如图8-11所示。然后会出现图8-12所示的画面。

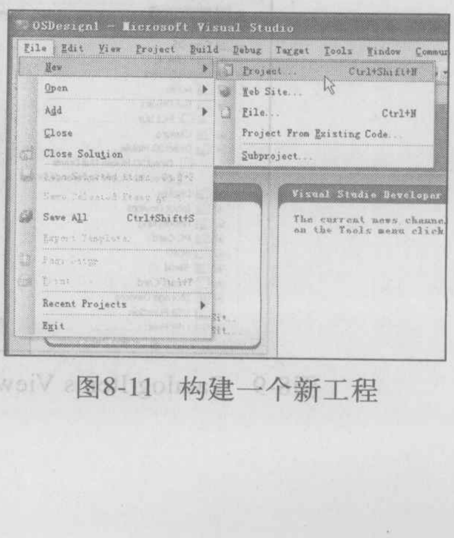


图8-11 构建一个新工程

图8-12　选择工程文件

在图8-12中，选择Platform Builder for CE 6.0，然后选择OS Design，可以在Name中输入想要的工程文件名称。在这里输入DeeguaEmulator，在Location中则可以设置定制的OS文件存放的位置，在这里选择了G盘的SourceTest文件夹。设置好以后，单击OK按钮。此时会出现图8-13所示的向导欢迎界面。

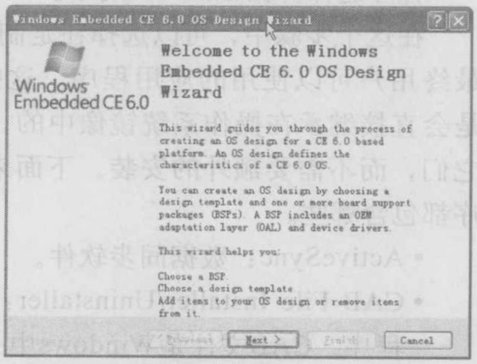


图8-13　向导欢迎界面

在图8-13所示的界面中，单击Next按钮，进入到BSP的选择界面中。

8.2.2　板级支持包BSP的选择

选择BSP的界面如图8-14所示。

在进行相应的选择操作以前，首先看一下什么是BSP。BSP的英文全称是Board Support Package，即板级支持包。它是介于主板硬件和操作系统之间的一层，也可以认为是属于操作系统的一部分。其主要作用是支持操作系统，使之可以更好地适应计算机的主板。一般地，BSP是相对于操作系统而言的，因此不同的操作系统对应于不同定义形式的BSP。例如，Windows CE的BSP和Linux的BSP相对于某一CPU来说尽管实现的功能一样，可是写法和接口定义则完全不同。纯粹的BSP所包含的内容一般来说是和系统相关的驱动或应用程序，如网络驱动程序、网络协议、串口驱动程序和下载调试等，离开这些驱动系统就不能正常工作了。

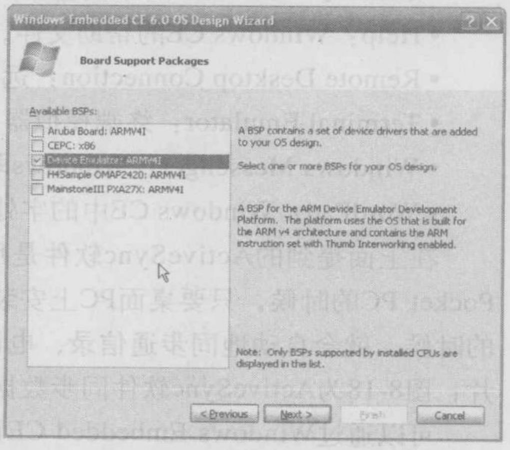


图8-14　BSP的选择

在Windows CE中，系统支持的处理器主要包括流行的X166系列、ARM系列、MIPS系列、

SH4系列等。除了包含每一系列中相应的标准开发包外，通常还带有该系列中常用的开发包。如在X86系列中，除了标准的CEPC开发包，还附带一个NATIONAL GEODE：X86开发包。

在这里需要特别提及的是，当在图8-14所示的界面中选择相应的BSP的时候，就会在界面右侧的信息栏中显示该BSP的相关信息。在这里选择Device Emulator：ARMV4I，即基于ARM体系结构的模拟器，然后单击Next按钮，进入到模板选择界面。

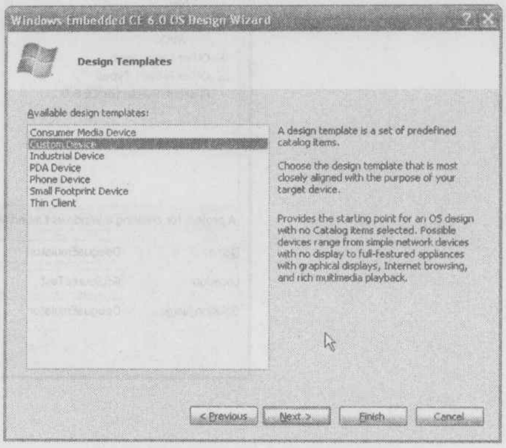


图8-15 模板选择界面

8.2.3 模板选择

模板选择界面如图8-15所示。

在这里选择Custom Device，即自定义设备。然后单击Next按钮进入到程序选择界面。

8.2.4 程序选择

程序选择界面如图8-16所示。

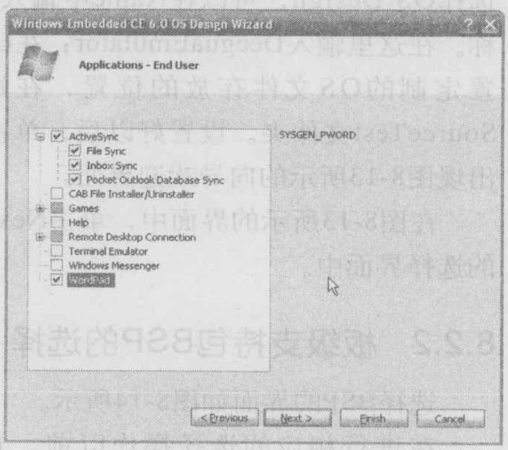


图8-16 程序选择界面

在这个步骤中，可以选择在定制的操作系统镜像中，最终用户可以使用的应用程序。这些程序被选择以后，是会直接继承在操作系统镜像中的，用户可以直接使用它们，而不需要额外的安装。下面来看一些这些应用程序都包含哪些。

- ActiveSync：数据同步软件。
- CAB File Installer/Uninstaller：CAB文件安装/卸载程序。CAB文件是Windows中一种高压缩率的压缩文件。原来只在Windows 98中出现过，现在则有很多的软件都采用了这种压缩方式。
- Games：Windows CE下面自带的一些游戏程序。
- Help：Windows CE的帮助文件。
- Remote Desktop Connection：远程桌面连接程序。
- Terminal Emulator：终端模拟器。
- Windows Messenger：Windows即时通信软件。
- WordPad：Windows CE中的字处理软件。

在上面提到的ActiveSync软件是用于PC和Windows CE设备之间通信的软件。例如，在使用Pocket PC的时候，只要桌面PC上安装了ActiveSync程序，当把Pocket PC和PC用数据线连接起来的时候，就会自动地同步通信录、电话本等信息。图8-17为ActiveSync软件同步数据时PC端的图片；图8-18为ActiveSync软件同步数据时Pocket PC端的图片。

可以通过Windows Embedded CE设备自带的红外线接口、蓝牙接口或者数据线等方式和PC进行数据通信。在使用无线通信方式的时候，不但可以免去数据线的烦琐，还可以大大地增强通信时候的便携性和可移动性。以蓝牙为例，在数据同步的时候，只需要保证Windows CE设备在有限距离之内即可，而无须考虑数据线和端口对应（使用红外连接的时候，PC端和Windows CE设

备端的红外端口需要保持一定角度的正对）的限制。

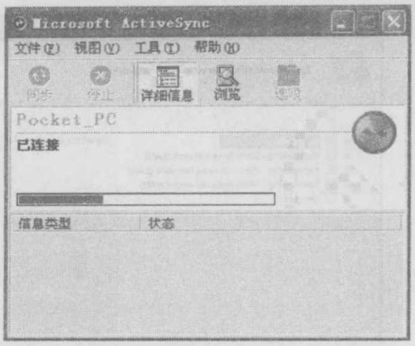


图8-17　同步数据时PC端的图片

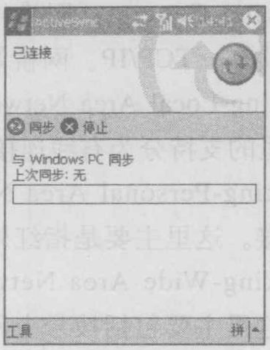


图8-18　同步数据时Pocket PC端图片

在这里勾选ActiceSync，然后单击Next按钮，进入到程序开发类库的选择界面。

8.2.5　程序开发类库的选择

程序开发类库的选择界面如图8-19所示。

在这个对话框中，可以选择基于定制的操作系统镜像所涉及的程序开发类库，它们主要包括（按照在列表中出现的顺序介绍）：

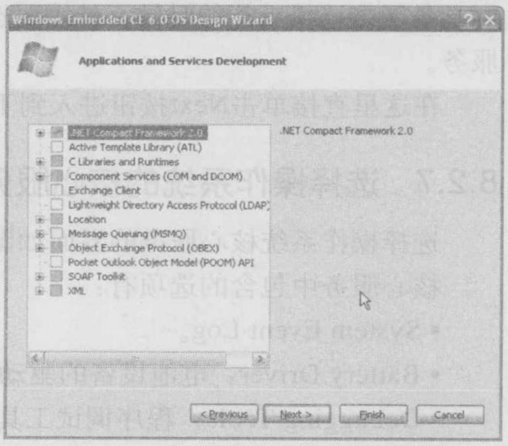


图8-19　程序开发类库的选择界面

- .NET Compact Framework 2.0：精简版.Net框架。
- Active Template Library（ATL）：活动模板库。
- C Libraries and Runtimes：C语言链接库。
- Component Services（COM and DCOM）：组件服务。
- Exchange Client。
- Lightweight Directory Access Protocol（LDAP）Client：客户端轻量目录访问协议。这种协议支持访问Internet所必须的TCP/IP。
- Location。
- Message Queuing（MSMQ）：消息队列。其全称为Microsoft Message Queue，简称为MSMQ。它是Windows NT的一项服务，可以在分布式程序间提供有保证的、事务性的和同步的消息通信。MSMQ 可以缩短开发电子商务系统的时间。
- Object Exchange Protocol（OBEX）：对象交换协议。
- Pocket Outlook Object Model(POOM)API。
- SOAP Toolkit：简单对象访问协议。它是一种用于再分散或分布式的环境中交换信息的简单的协议，也是一个基于XML的协议。
- XML。

同样可以根据实际需要进行相应的选择，由于在这里主要是展示整个内核定制的过程，因此该步骤中没有勾选任何选项，直接单击Next按钮进入到下一个界面。

8.2.6　通信服务和网络连接

通信服务和网络连接的界面如图8-20所示。

在这个步骤中，主要任务是选择操作系统镜像的连接服务和网络服务。主要包括：

- Networking-General：网络特性。这里主要包括网络连接共享、TCP/IP、网桥等信息。
- Networking-Local Area Network（LAN）：局域网连接。这里的支持分为有线连接和无线连接。
- Networking-Personal Area Network（PAN）：个人网络连接。这里主要是指红外线连接等。
- Networking-Wide Area Network（WAN）：广域网连接。这里主要包括拨号上网等连接。
- Servers：服务。这里主要包括FTP服务等。

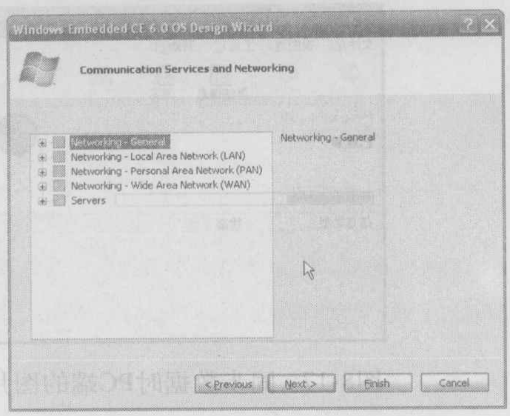


图8-20 通信服务和网络连接

在这一步骤中，可以根据不同的嵌入式产品的特点，选择不同的网络服务。例如，当订制的操作系统镜像需要具备网络连接功能的时候，就需要选择Networking-Wide Area Network（WAN）广域网连接等服务。

在这里直接单击Next按钮进入到下一个界面。

8.2.7 选择操作系统的核心服务

选择操作系统核心服务的界面如图8-21所示。

核心服务中包含的选项有：

- System Event Log。
- Battery Driver：电池设备的驱动。
- Debugging Tools：程序调试工具。
- Device Manager。
- Display Support。
- Internet Application (IABASE) Support。
- Kernel Functionality：内核函数。
- Notification (IABASE)。
- Notification LED Support。
- Parallel Port Support：对并口的支持。
- Power Management (IABASE)：电源设备管理。
- Serial Port Support：对串口的支持。
- UI Proxy for Kernel-Mode Drivers。
- USB Host Support：对USB的支持。
- Windows Embedded CE Driver Development Kit Support Library。

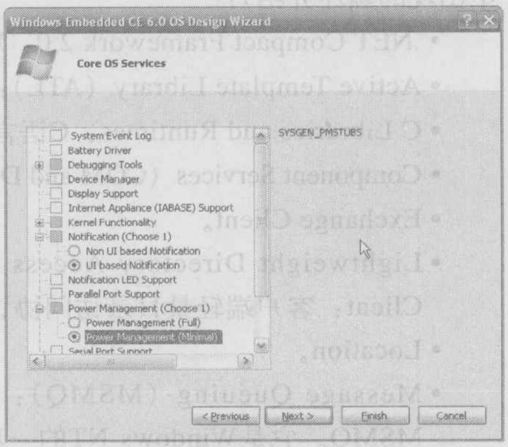


图8-21 选择操作系统的核心服务

在上面提到的串口通常也被叫做串行接口。串行口不同于并行口之处在于它的数据和控制信息是一位接一位地传送出去的。虽然这样速度会慢一些，但传送距离较并行口更长，因此若要进行较长距离的通信，应使用串行口。而并口通常又称为并行接口。并行接口通常作为打印机端口。和串口不同，并口可以把8位数据同时进行传送，这样数据传送速度大大提高。但是并行传送受

到了传输距离长短的限制，因为长度增加，干扰就会增加，数据也就容易出错。

图8-22为计算机的串口，图8-23为计算机的并口。

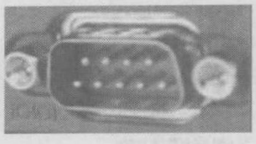


图8-22 计算机的串口

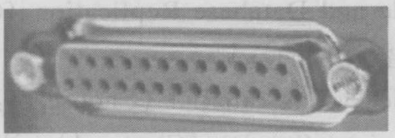


图8-23 计算机的并口

Windows Embedded CE作为一款嵌入式操作系统，自然具备了许多嵌入式设备的一些特性。电源管理（Power Management）就是其中之一。下面来简单地看一下，Windows Embedded CE中的电源管理具有哪些特点。

在Windows Embedded CE中，系统是通过电源管理器来进行电源管理的。电源管理器可以提高整个Windows CE系统的电源利用率，减少嵌入式设备的电能消耗。同时还可以在嵌入式设备重启、运行或者挂起的状态下保存在RAM中的文件系统。

Windows Embedded CE中的电源管理器主要作用为：

- 可以枚举电源管理的具体功能。
- 可以处理和电源管理相关的服务请求。
- 可以在Windows CE系统重新启动以后迅速地给设备供电。
- 支持嵌入式设备的休眠、唤醒等服务。

例如，当使用嵌入式设备播放音乐的时候，需要保持声音解码设备和声卡处于较高级别的电源等级上。在进行视频欣赏的时候，还需要额外地保持显卡处于高级别的电源等级上。当系统进入到休眠状态的时候，则要求电源处于较低级别的电源等级。经常使用笔记本电脑的朋友一定会有这样的体会，当笔记本电脑采用交流电供电的时候，CPU和屏幕的亮度通常会处于全速和高亮状态。而当采用笔记本电池供电的时候，屏幕的亮度则会变暗，CPU的速度也会降低。正是通过这种方式，使得嵌入式移动设备可以处于长时间的工作状态。要知道，电池使用时间的长短对于嵌入式设备来说，起着举足轻重的作用。

在这个步骤中，选择Display Support和UI Proxy for Kernel-Mode Drivers，然后单击Next按钮进入到下一个界面。

8.2.8 设备管理

设备管理的界面如图8-24所示。

在这里直接单击Next按钮，进入到文件系统和数据存储方式选择界面。

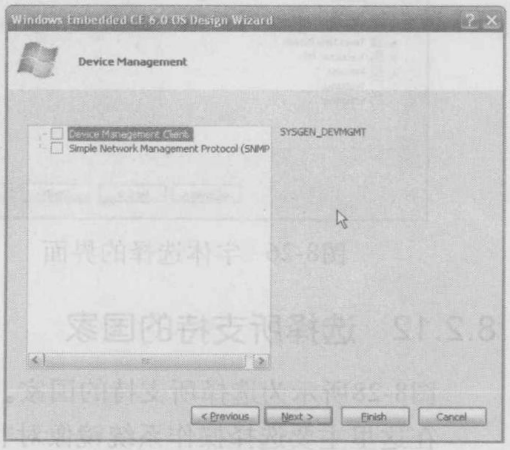


图8-24 设备管理界面

8.2.9 选择文件系统和数据存储方式

文件系统和数据存储方式选择界面如图8-25所示。

在这一步中，主要工作是选择操作系统镜像中的文件系统和数据存储方式。主要包括：

- Compression。
- Database Support：数据库支持。
- File and Database Replication（Choose 1）：文件与数据库的复制。
- File System-Internal（Choose 1）：内部文件系统。
- Registry Storage（Choose 1）：注册表存储。
- Storage Manager：存储管理。
- System Password：系统密码。

在这里选择RAM and ROM File System和RAM-based Registry，然后单击Next按钮进入到下一个界面。

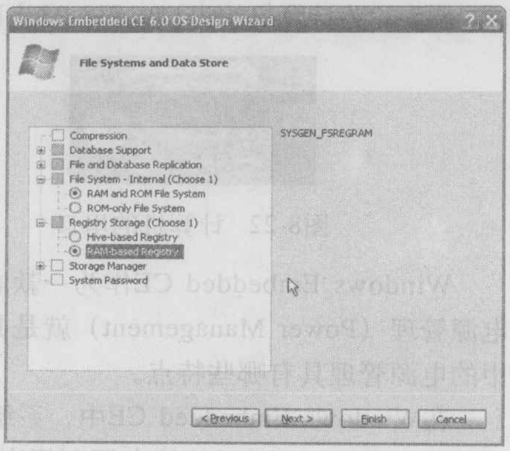


图8-25 文件系统和数据存储方式选择界面

8.2.10 字体的选择

字体选择的界面如图8-26所示。

在这里需要选择操作系统镜像所支持的字体。在选择相应的字体以后，单击Next按钮，进入到下一个界面。

8.2.11 图形和多媒体技术的支持

其界面如图8-27所示。

在这里直接单击Next按钮进入到下一个界面。

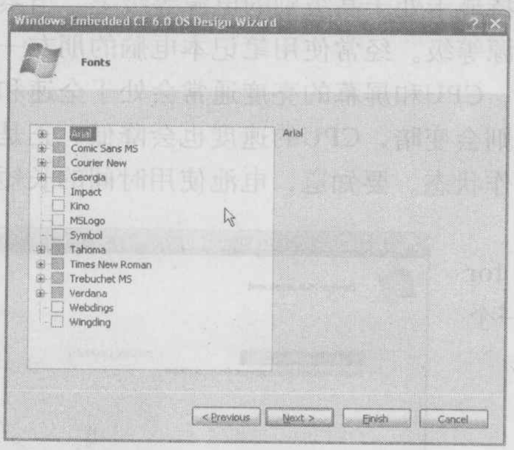


图8-26 字体选择的界面

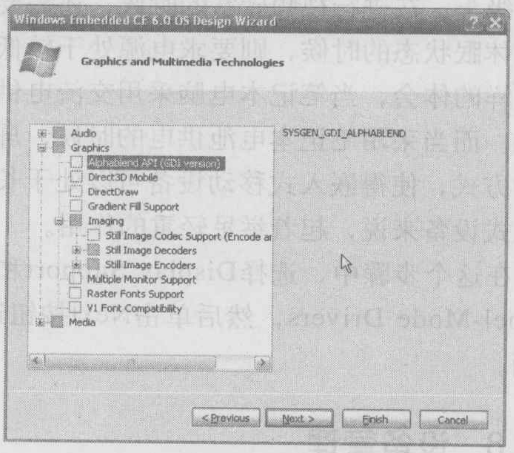


图8-27 图形和多媒体的支持

8.2.12 选择所支持的国家

图8-28所示为选择所支持的国家。

在这里主要选择操作系统镜像对非英语国家的支持，如输入法、语言等。具体内容主要包括：

- Input Method Manager（IMM）。
- Local Services（Choose 1）：地区服务。在这个选项下面，又具有两个子选项。
- Local Specific Support：具体地区的支持。在这里也包含了很多具体的选项，如中文简体、

中文繁体、英语、法语等。

- Multilingual User Interface（MUI）：多语言用户界面。
- Unicode Script Processor for Complex Scripts。

图8-29展示了各个选项中的子项。

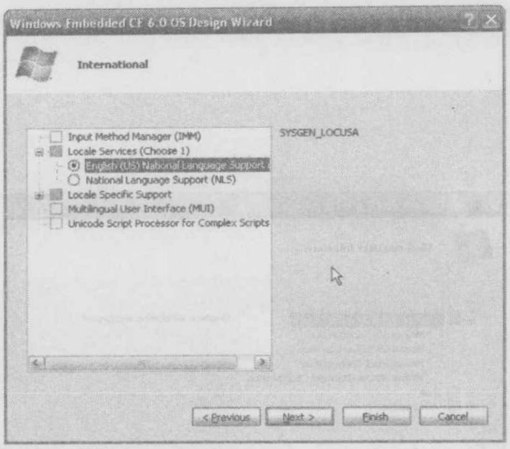


图8-28　选择所支持的国家

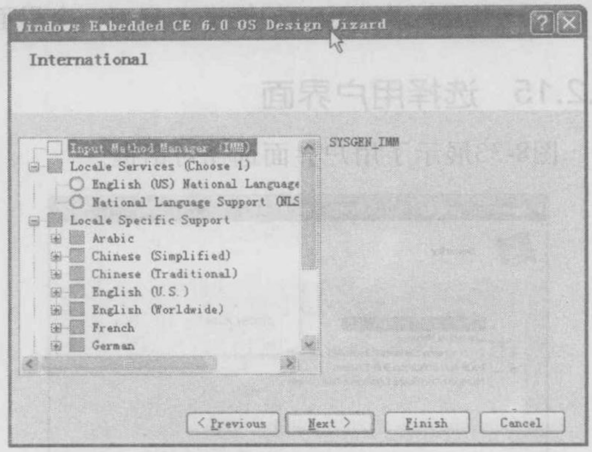


图8-29　具体的子项

在这里直接单击Next按钮进入到下一个界面。

8.2.13　选择客户端的Internet服务

图8-30展示了选择客户端Internet服务的对话框。

在这一个步骤中，主要工作是选择和Internet服务相关的选项。这里面主要包括：

- Browser Application：网络浏览器。
- Internet Explorer 6.0 for Windows Embedded CE Components：Windows Embedded CE中用的IE6浏览器组件。
- Internet Options Control Panel：IE6控制面板。
- Scripting：脚本语言。

图8-31展示了本步骤中的一些子选项。

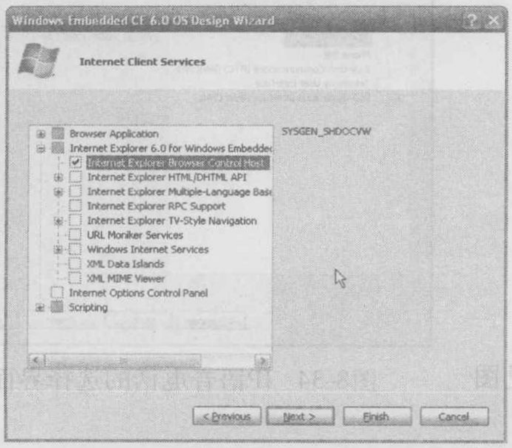


图8-30　选择客户端Internet服务

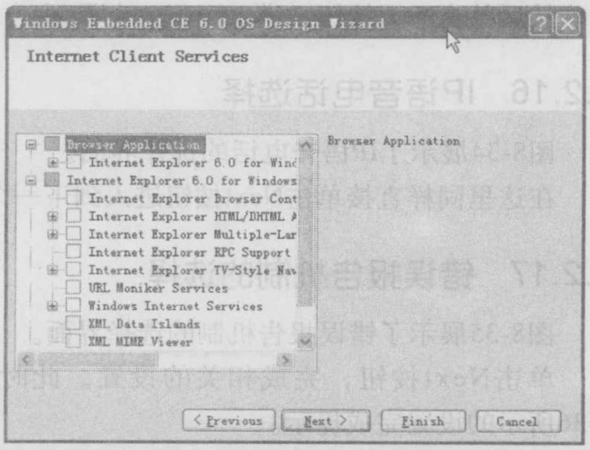


图8-31　Internet服务中的一些子选项

在这里直接单击Next按钮进入到下一个界面。

8.2.14 安全属性的选择

安全属性的选择对话框如图8-32所示。

在这里直接单击Next按钮进入到下一个界面。

8.2.15 选择用户界面

图8-33展示了用户界面选择对话框。

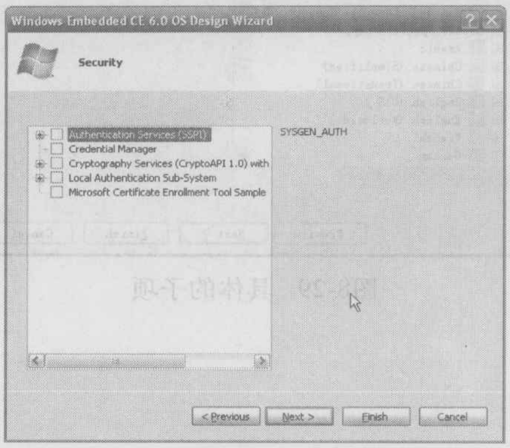


图8-32 安全属性的选择对话框

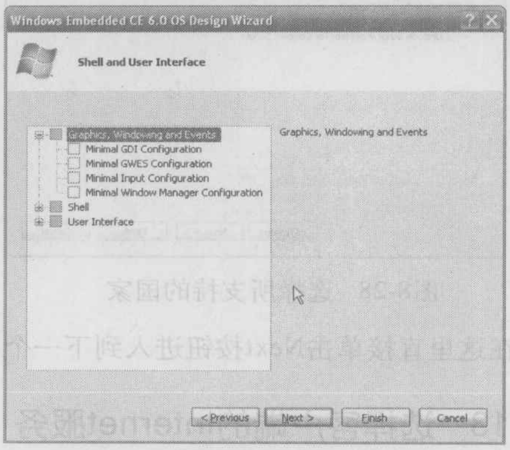


图8-33 用户界面选择对话框

在这里的主要工作是选择与定制的操作系统镜像相关的用户界面等信息。在这里主要包括：

- Graphics，Windowing and Events：图像、窗口和事件。
- Shell：此功能主要包括操作系统镜像的控制面板等系统的风格。
- User Interface：用户界面，主要包括触摸屏的显示风格等信息。

在这里需要选择以下两个选项：

- Minimal GDI Configuration。
- Minimal GWES Configuration。

然后单击Next按钮，进入到下一个界面。

8.2.16 IP语音电话选择

图8-34展示了IP语音电话的选择界面。

在这里同样直接单击Next按钮进入到下一个界面。

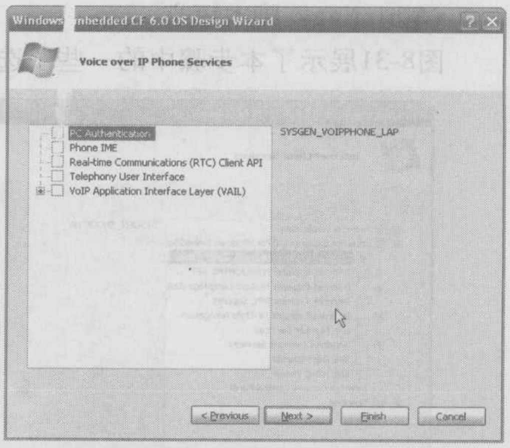


图8-34 IP语音电话的选择界面

8.2.17 错误报告机制的选择

图8-35展示了错误报告机制的选择界面。

单击Next按钮，完成相关的设置。此时会出现图8-36所示的设置完成提示。

单击Finish按钮，完成整个操作系统的设置工作。

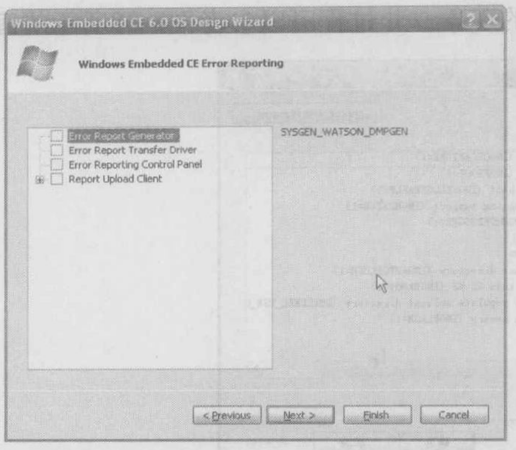


图8-35 错误报告机制的选择界面

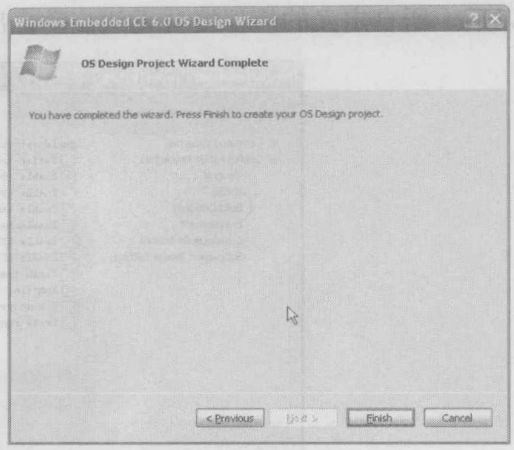


图8-36 设置完成

8.3 生成操作系统镜像

在本章前面的内容中，已经介绍了使用Platform Builder for CE 6.0建立一个操作系统镜像的方法，接下来要做的工作就是对这个建立的镜像文件进行编译，以生成所需要的操作系统镜像。

在Platform Builder for CE 6.0中，支持两种镜像文件。其中一种被称为Release版本，一种被称为Debug版本。这里提到的Release版本主要是交付给用户使用的版本，而Debug版本则是用于调试的版本。既然作为调试版本，Debug版本中自然包含了很多和调试相关的信息。在Debug版本中，系统自动地作了很多工作，如内存管理等。但是Release版本则相对简化。因此有的时候，当程序在Debug版本调试通过，并且可以正确执行的时候，并不一定表示Release版本一定可以被正确地执行。这就需要小心地编写程序。

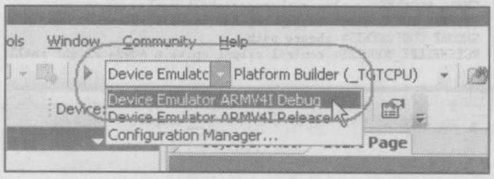


图8-37 两种可以选择的版本

图8-37展示了Platform Builder for CE 6.0中两种可以选择的版本。

在这里选择了Debug版本来进行编译，这个过程可以参考图8-37所示的画面。需要特别指出的是，在这里选择的版本和后面镜像sdk的编译直接相关，即Debug版本的镜像文件必须和Debug版本的SDK相对应，Release版本的镜像文件必须和Release版本的SDK相对应。具体操作说明可以参考本章后面有关SDK的内容介绍。

选择好编译的版本以后，接下来要做的就是设置编译时候的一些与镜像文件相关的参数。具体操作为单击Visual Studio 2005工具栏中的Project，然后选择里面的Properties，此时会出现图8-38所示的画面。

在图8-38中列出了与编译相关的一些参数设置。在这里首先选择左侧的Build Options选项，然后会在Build options中看到有以下三项被选择：

- Enable eboot space in memory (IMGEBOOT=1)。
- Enable kernel debugger(no IMGNODEBUGGER=1)。
- Enable KITL (no IMGNOKITL=1)。

在这里需要把第3项Enable KITL (no IMGNOKITL=1)去掉，否则导出的SDK在安装到目标设

备上以后，所自定义的模拟器会无法启动，出现黑屏的现象。此过程可以参考图8-38所示的画面。

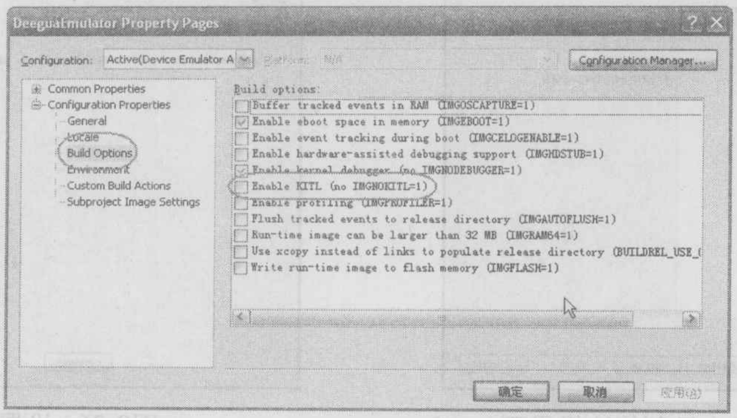


图8-38 编译参数的选择

设置好以后，单击“确定”按钮，完成相关的参数设置。此时可以在工程文件的上面单击鼠标右键，选择Build Solution来进行镜像文件的编译，如图8-39所示。

在镜像文件编译的过程中，可以在Visual Studio 2005的Output窗口中看到正在编译的内容，如图8-40所示。

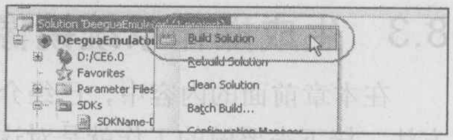


图8-39 Build Solution

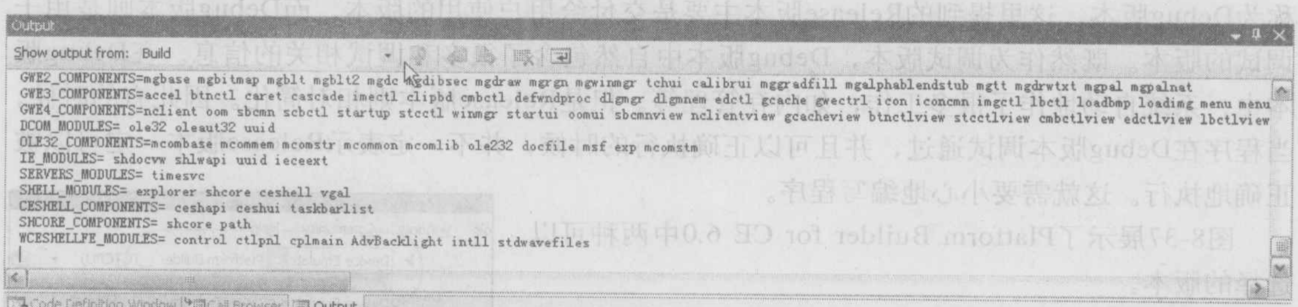


图8-40 Build过程

编译好以后会出现图8-41所示的画面。

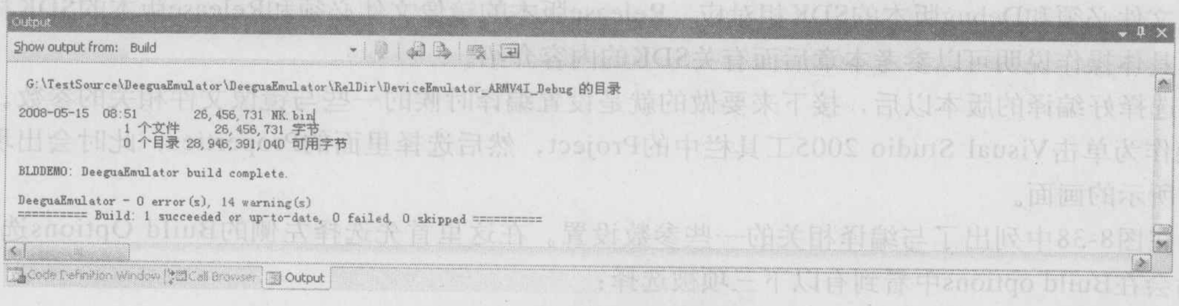


图8-41 镜像文件编译成功

8.4 运行生成的操作系统镜像

镜像编译成功以后，接下来要做的就是运行所编译的镜像文件。首先单击Visual Studio 2005

工具栏上的Target，选择里面的Connectivity Options，如图8-42所示。

上述操作完成以后，会出现图8-43所示的画面。在图8-43所示的画面中需要进行如下设置：

• Download：选择Device Emulator （DMA）。
• Transport：选择Device Emulator（DMA）。
• Debugger：选择KdStub。

上述参数设置好以后，单击Download后面的Settings按钮，此时会出现图8-44所示的画面。

在图8-44中选择“显示”，然后在出现的对话框中把“视频”选项卡上的屏幕宽度设置成640，屏幕高度设置成480，如图8-45所示。

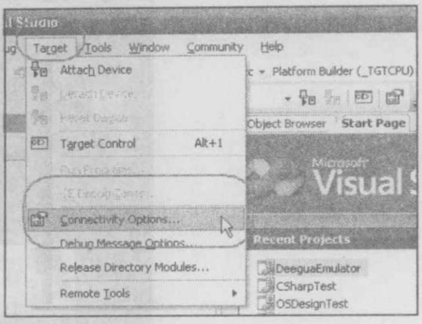
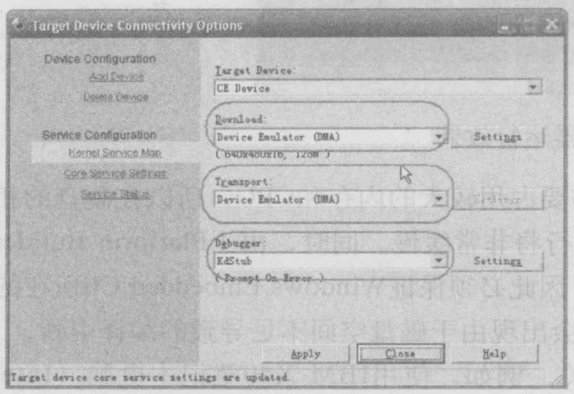


图8-42 配置连接参数



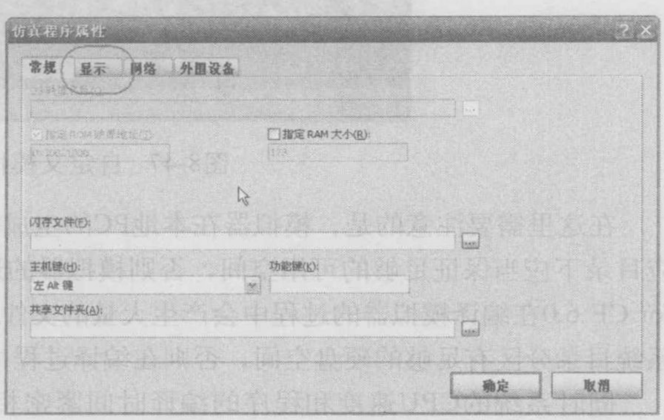
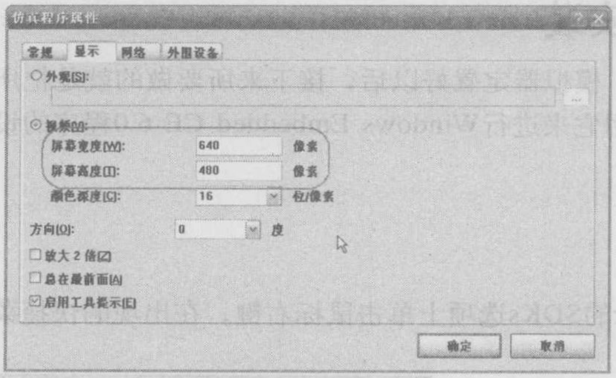


图8-43 连接选项设置　　　　图8-44 仿真程序属性

设置好以后，单击“外围设备”，勾选“为串行端口1创建文本控制台窗口”，如图8-46所示。



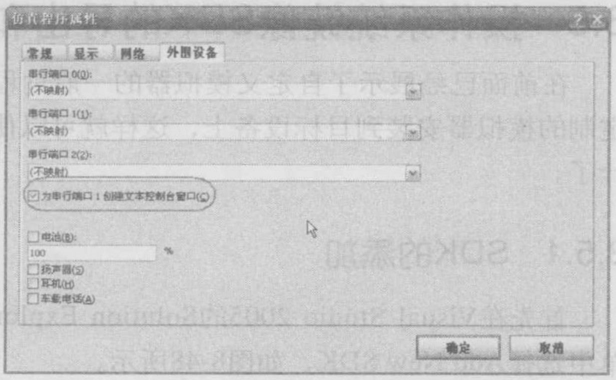


图8-45 仿真程序属性　　　　图8-46 外围设备参数设置

在这里之所以勾选这个选项，其作用就是在模拟器运行的时候可以通过屏幕参数的显示来查看内核调用的过程，便于发现问题。选择好以后，单击“确定”按钮，这样就完成了整个参数的设置。

在Visual Studio 2005的选项卡上选择Target，然后选择Attach Device，就可以看到自定义模拟器的效果了。图8-47展示了自定义模拟器运行后的效果。

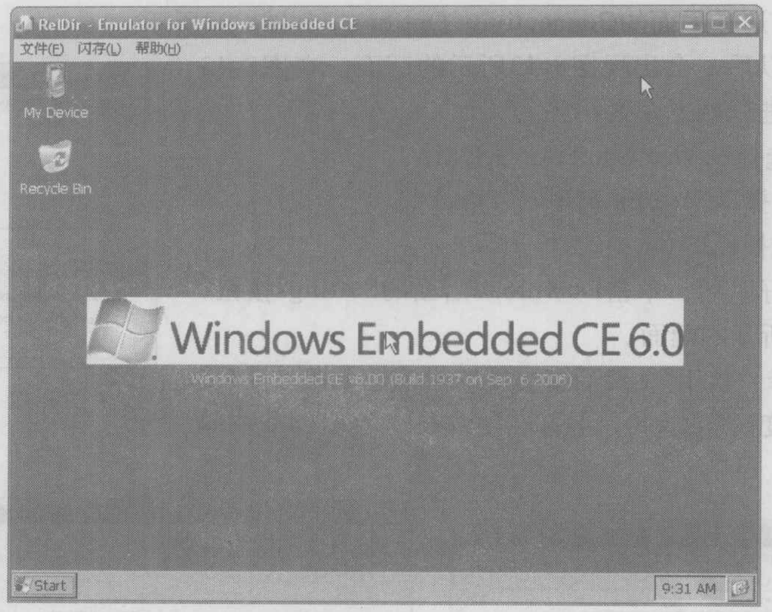


图8-47 自定义模拟器运行效果

在这里需要注意的是，模拟器在本地PC的生成需要占用较大的内存和CPU，所以在硬盘的相应目录下应当保证足够的可用空间，否则模拟器的运行将非常缓慢。同时，由于Platform Builder for CE 6.0在编译模拟器的过程中会产生大量的文件，因此必须保证Windows Embedded CE所在的系统目录分区有足够的硬盘空间，否则在编译过程中会出现由于磁盘空间不足导致的编译中断。

同时系统的CPU速度和程序的编译时间紧密相关。例如，使用IBM X40笔记本电脑（Intel Pentium M 1.60 GHz CPU）编译时大概需要40分钟的时间，而使用Intel酷睿2 E8300双核处理器（2.86GHz CPU），大概只需要10分钟的时间。

8.5 操作系统镜像SDK的导出和安装

在前面已经展示了自定义模拟器的一般过程。模拟器定置好以后，接下来所要做的就是把所定制的模拟器安装到目标设备上，这样就可以使用它来进行Windows Embedded CE 6.0程序的设计了。

8.5.1 SDK的添加

首先在Visual Studio 2005的Solution Explorer的SDKs选项上单击鼠标右键，在出现的快捷菜单中选择Add New SDK，如图8-48所示。

在出现的图8-49所示的菜单中，可以根据需要设置与SDK相关的一些参数。在这里把相关参数设置为：

- SDK Name：SDKName-DeeguaEmulator。
- Product Name：PDName-Deegua。
- Company Name：deegua。
- Company Website：www.deegua.com。

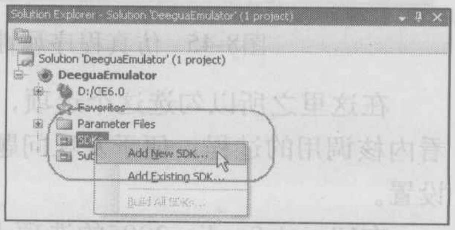


图8-48 添加新的SDK

此过程可以参考图8-49所示的画面。

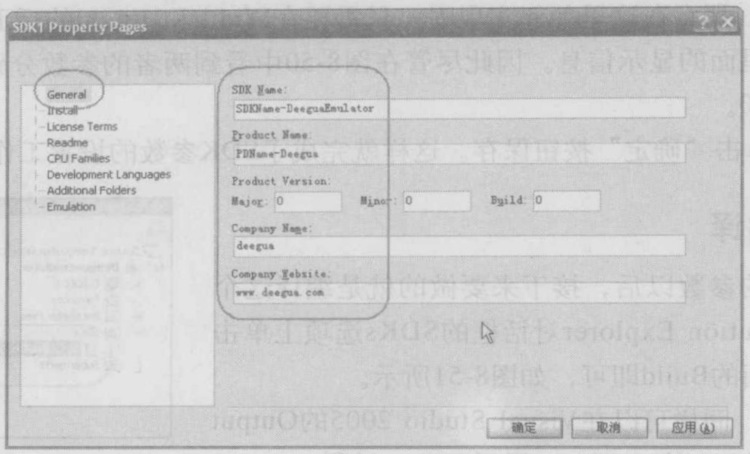


图8-49 SDK General属性设置

设置好SDK名称、产品名称等参数以后，单击左侧的Emulation选项，此时会出现图8-50所示的画面。在Emulator选项中需要对SDK安装以后所运行的模拟器参数进行相应的设置。需要特别提及的是，系统默认情况下导出的SDK是不包含模拟器的，因此要想在自定义模拟器上运行程序，就必须进行相应的设置。具体设置参数为：

- Configuration：选择Device Emulator ARMV4I Debug。
- Fixed screen resolution：
  - Width：640。
  - Height：480。

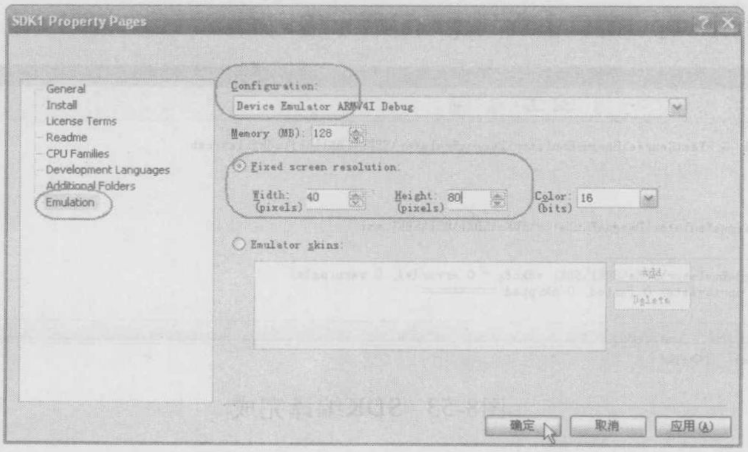


图8-50 SDK Emulator 参数的设置

细心的读者一定会发现，在Configuration选项中，还可以选择Release版本，但是在这里必须和前面的编译版本所对应，这也就是在本章前面特别提到的注意编译版本的原因所在。如果感兴趣，不妨在这里选择Release版本尝试一下，看看会有什么情况发生。

Width和Height选项分别设置了模拟器的宽度和高度。既然涉及到这里，就不得不提Visual Studio 2005的一个bug。如果直接单击Width或者Height边上的“上下”按钮进行参数设置，会发

现只可以把宽和高设置在0~99的范围内。但是实际的Windows Embedded CE 6.0设备屏幕分辨率一般都为640 × 480，那么在这里如何实现呢？只需要在Width和Height的后面直接输入640和480即可，而不用去管里面的显示信息。因此尽管在图8-50中看到两者的参数分别为40和80，但是实际上应该是640和480。

设置好以后，单击“确定”按钮保存。这样就完成了SDK参数的设置工作。

8.5.2 SDK的编译

在设置好SDK的参数以后，接下来要做的就是编译这个SDK。只需要在Solution Explorer对话框的SDKs选项上单击鼠标右键，选择里面的Build即可，如图8-51所示。

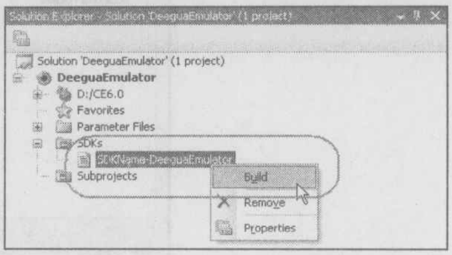


图8-51 Build SDK

在Build过程中，同样可以在Visual Studio 2005的Output窗口中看到相应的编译过程。图8-52展示了这一过程。

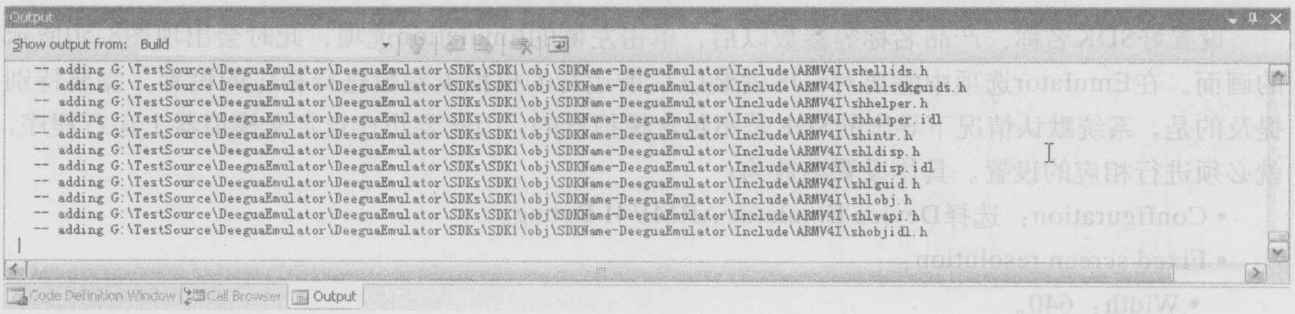


图8-52 SDK Build过程

编译好以后，会在Output窗口中看到图8-53所示的画面。

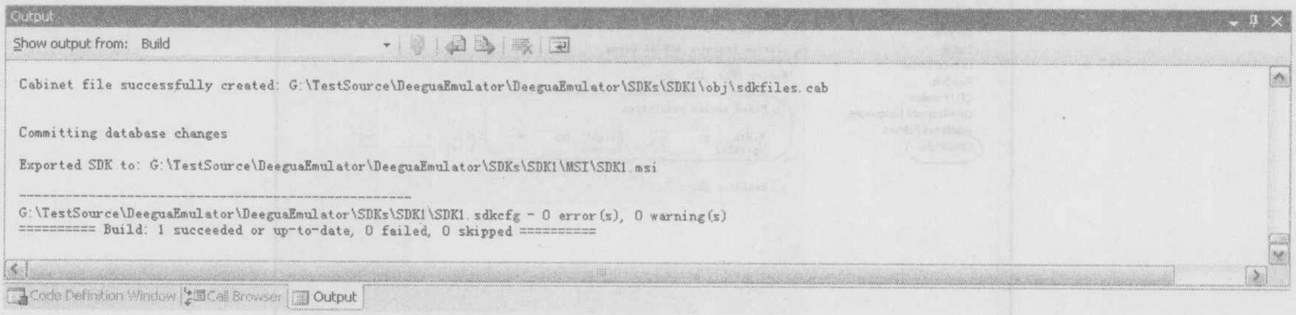


图8-53 SDK编译完成

8.5.3 SDK的安装

SDK编译好以后，接下来要做的事情就是安装SDK。首先找到SDK安装文件所在的目录，以笔者的电脑为例，SDK所在的目录为G:\TestSource\DeeguaEmulator\DeeguaEmulator\SDKs\SDK1\MSI，如图8-54所示。

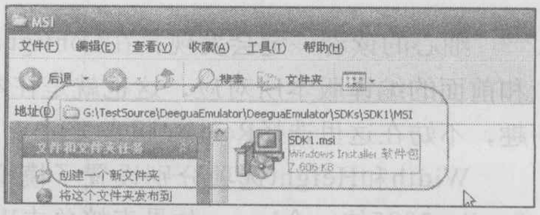


图8-54 SDK文件所在的目录

双击文件进行安装，安装好以后就可以在Visual Studio 2005中看到自定义的模拟器。图8-55展示了这个过程。

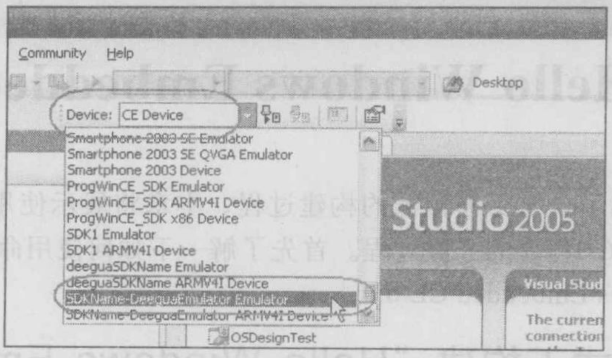


图8-55 在Visual Studio 2005中选择自定义的模拟器

第9章　Hello Windows Embedded CE 6.0

在上一章中已经介绍了自定义模拟器的构建过程，本章将展示使用Visual Studio 2005编写Windows Embedded CE 6.0程序的一般过程。首先了解一下如何使用命令行模式在Visual Studio 2005中显示Hello Windows Embedded CE 6.0。

9.1 在命令行模式下构建“Hello Windows Embedded CE 6.0程序”

Visual Studio 2005是微软公司新推出的一个集成开发环境，在这里面包含了Visual C#、Visual C++、Visual Basic等开发工具，同时新版本的Platform Builder for CE 6.0也作为一个插件出现在Visual Studio 2005中。本书会分别介绍使用Visual C#、Visual Basic等进行Windows Embedded CE 6.0程序设计的过程。首先来看一下如何在命令行模式下，使用Visual C# 2005显示Hello Windows Embedded CE 6.0。

Visual C# 2005是Visual Studio 2005中非常重要的开发工具，它包括交互式开发环境、可视化设计器、编译器和调试器等。Visual C# 2005是 Visual Studio 2005产品套件的组成部分，该产品套件还包括 Visual Basic 2005、Visual C++ 2005 和 JScript 脚本语言。所有这些语言都支持对Microsoft .NET Framework 的访问，因此，对开发人员而言，虽然C#是一种新的语言，本身不包含类库，但仍然可以使用其他如VB 和C++等成熟工具，使用相同的、功能丰富的类库。

下面将通过一个简单的例子来看一下如何显示Hello Windows Embedded CE 6.0。程序代码为：

```
using System;
class HelloWinCE
{
    public static void Main()
    {
        Console.WriteLine("Hello Windows Embedded CE!");
    }
}
```

可以把上面这段代码放在一个文本文件中，然后把文本文件的名称更改为Hello Wince .cs。其中的“.cs”是文件的扩展名，是C Sharp的简写。接下来，需要把这个文件放在Visual Studio 2005的安装目录下，如把该文件放在D:\VS2005\VC中。然后在Visual Studio Tools（在开始菜单的Visual Studio 2005选项中可以找到）中选择Visual Studio 2005 Command Prompt，此时会出现图9-1所示的对话框。

在出现的对话框中，输入下面的命令对HelloWince.cs文件进行编译：

```
csc HelloWince.cs
```

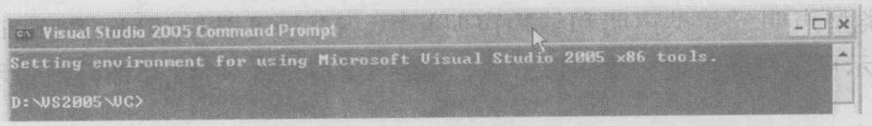


图9-1 Visual Studio 2005命令行窗口

此时会出现图9-2所示的对话框。

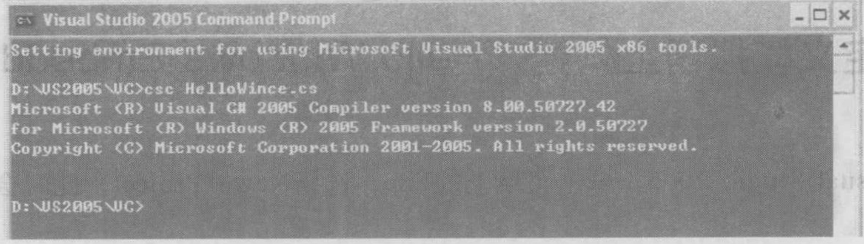


图9-2 编译HelloWince文件

然后直接输入HelloWince即可，此时窗口中会显示出来“Hello Windows Embedded CE！”字样，如图9-3所示。

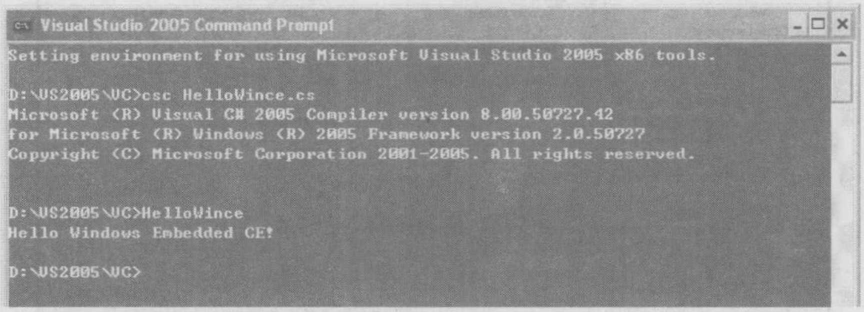


图9-3 显示Hello Windows Embedded CE 6.0！

在这里需要指出的是，前面输入的命令csc表示C Sharp Compiler，也就是编译相应的C Sharp文件。当然，可以通过Visual Studio 2005集成开发环境来进行该程序的编译。关于采用Visual Studio 2005集成开发环境来进行编译的方法可参考本书后面程序实例的相关介绍。

熟悉Java程序设计的读者会发现，C#与Java有一些相似的地方，但有些又与Java或C++不同，可以通过上面的程序来简单了解一下C#相关信息：

首先，“using System;”指令引用一个名为System的命名空间，这是驻留最常用的.NET类的地方。这非常类似于Java中的import语句和C++中的using名称空间语句，但是，它与#include指令完全不同，因为在编译好的C#文件中没有包含任何代码，using指令只是告诉编译器到哪里查找不在当前名称空间中和没有指定名称空间的类。

其次，在该程序中，使用了一些符号：每行语句必须用分号结尾，并且用花括号把语句结合为块；使用多行代码写语句，不需要使用像VB中的下划线这样的续行字符，当出现有多行注释的，则以一个斜杠和一个星号开头，以一个星号和一个斜杠结尾（/*……*/）；分隔符“.”在符合名称中使用，如Console.WriteLine。

同时在C#中，Main是这个程序的入口点，它包含了该程序的实际代码，当程序开始运行时首先调用该方法。该程序中的Main方法不是全局的，C#不支持全局级别的方法和变量，这类元

素总是包含在类型声明（如类声明和结构声明）中。Main可以用public和static来修饰。其中public修饰符表示可以在任何地方使用该方法，包括类的外部；static修饰符则表示Main方法不能被实例化。

最后，该程序没有使用#include导入程序文本。程序间的依赖项通过符号而不是文本来控制，这样就消除了由多种语言编写的应用程序间的障碍。

接下来，一起学习一下如何在自定义模拟器上构建“Hello Windows Embedded CE 6.0”程序。

9.2 在自定义模拟器上构建“Hello Windows Embedded CE 6.0程序”

首先启动Visual Studio 2005，单击菜单上的File，选择New→Project，此时会出现图9-4所示的画面。

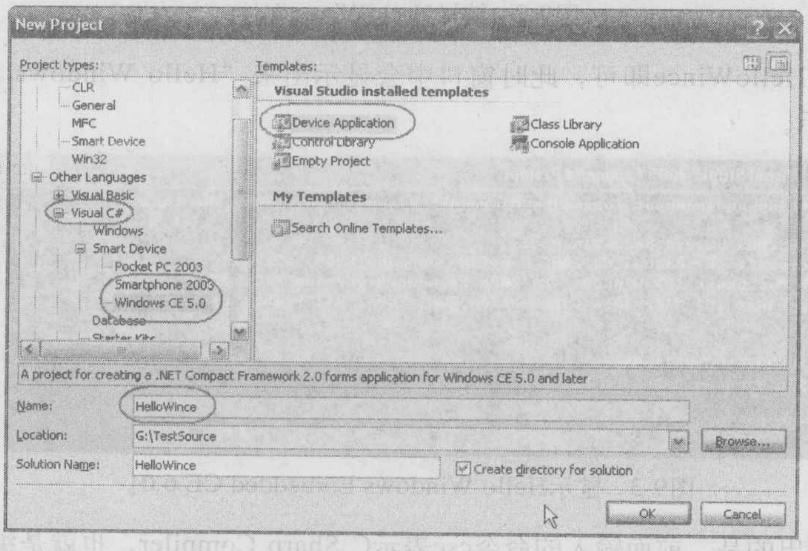


图9-4 New Project

在图9-4所示的画面中，在Other Languages中选择Visual C#，在Smart Device中选择Windows CE 5.0，在Visual Studio installed templates中选择Device Application。然后在Name中输入工程文件的名称，在这里输入HelloWince，设置好以后单击OK按钮完成工程文件的构建。

图9-5展示了构建好以后的工程文件。

在图9-5左侧的Solution Explorer中列出了工程所包含的文件，在这里可以查看程序的源代码等信息；在右侧Form1.cs中则展示了程序的用户界面。

接下来从右侧的Toolbox里面拖拽一个Button控件到Form1窗口中，如图9-6所示。

然后在Button1上双击鼠标左键，进入到代码编辑界面，图9-7展示了代码编辑界面。

在代码编辑界面中，需要在光标闪烁的位置输入下面一行代码：

```
MessageBox.Show("Hello Windows Embedded CE 6.0!")
```

这里面的MessageBox表示一个消息对话框；Show表示把这个消息对话框显示出来，括号中的“Hello Windows Embedded CE 6.0！”则展示了在消息对话框中需要显示的内容。通过这样一个简单的语句，就实现了在窗口中单击Button按钮，然后显示出一个对话框的简单功能。有关C#

程序设计更为复杂的内容，可参考本书后面的相关章节。

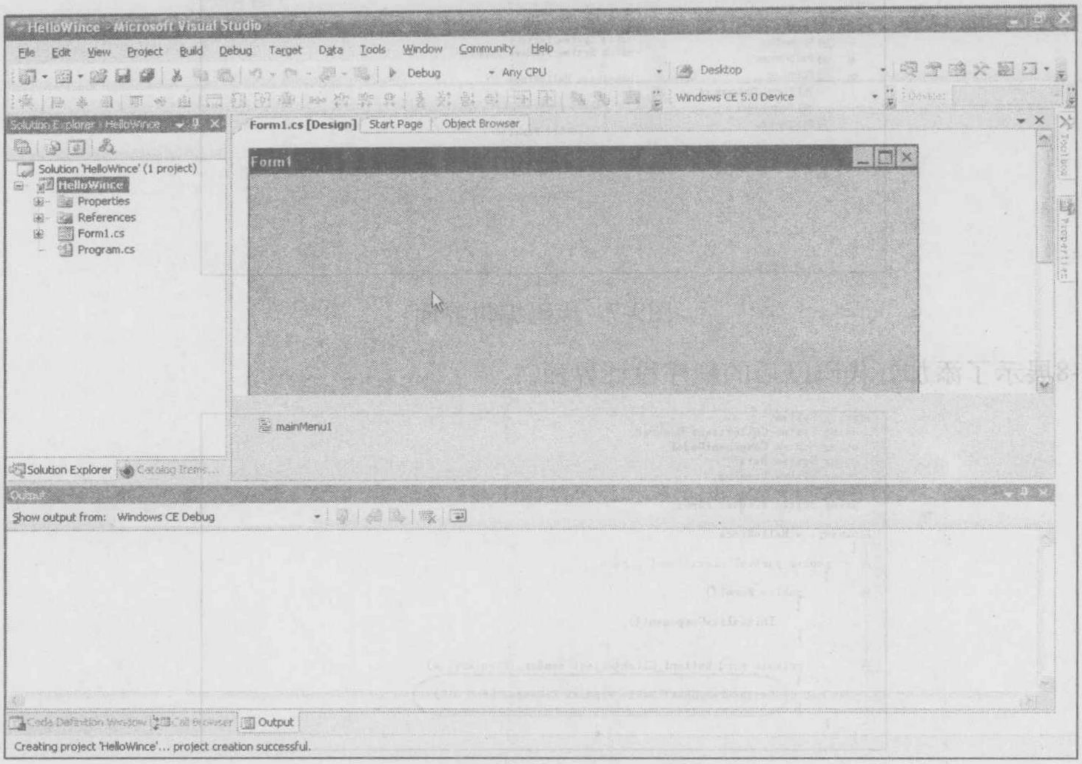


图9-5 构建好以后的工程文件

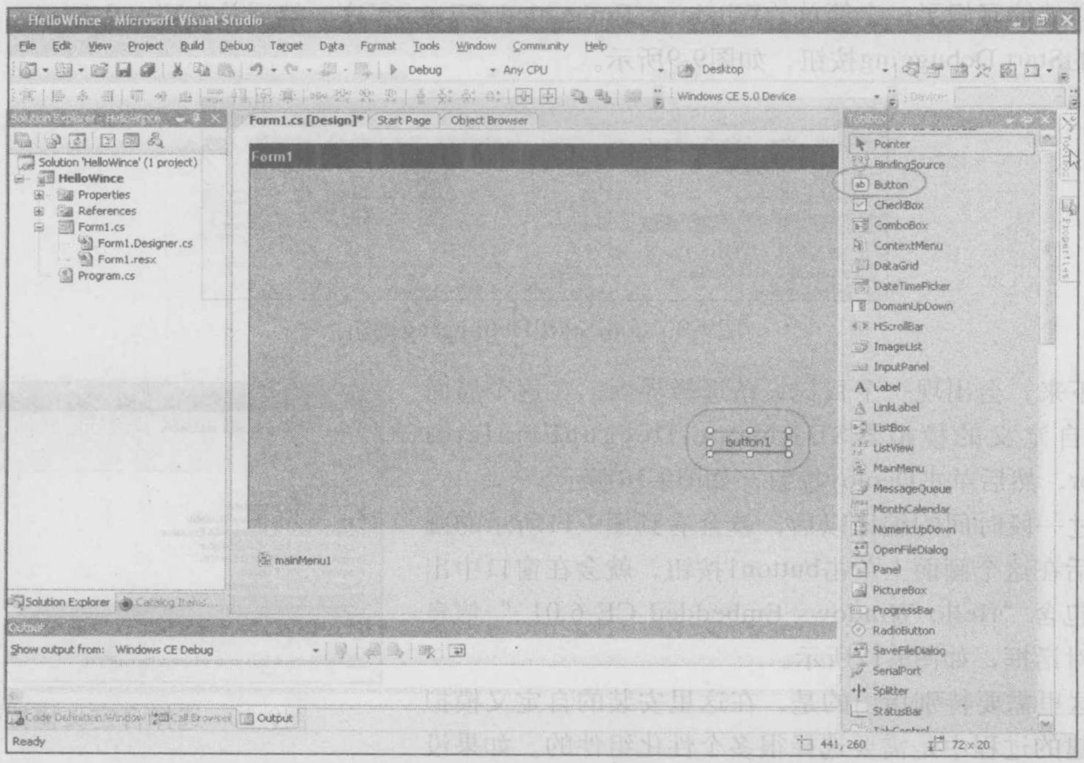


图9-6 拖拽一个Button控件到Form1窗口中

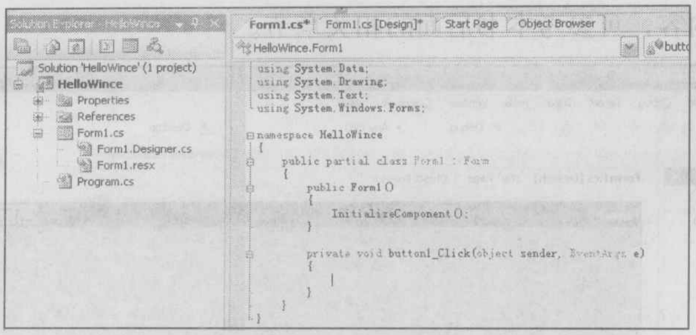


图9-7 代码编辑界面

图9-8展示了添加好代码以后的程序设计界面。

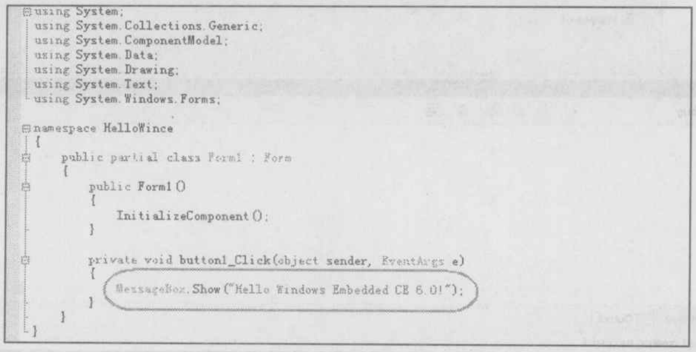


图9-8 输入所需要的代码

这样就编写好了一个简单的Windows Embedded CE 6.0程序。然后单击Visual Studio 2005工具栏上的Start Debugging按钮，如图9-9所示。

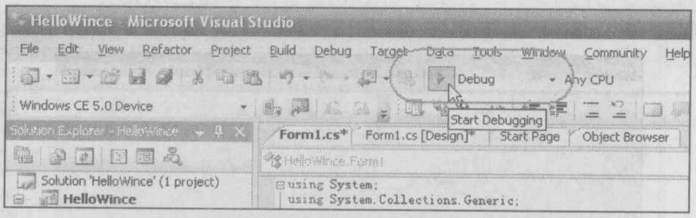


图9-9 单击Start Debugging按钮

接下来，会出现一个目标设备选择界面，在这个界面中选择自定义的模拟器SDKName-DeeguaEmulator Emulator，然后单击Deploy按钮，如图9-10所示。

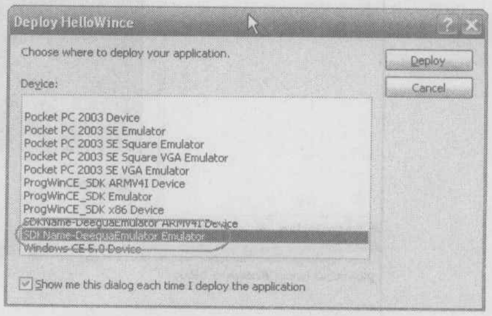


图9-10 选择自定义模拟器

经过一段时间的编译以后，就会看到图9-11所示的画面。然后在这个画面上单击button1按钮，就会在窗口中出现一个包含"Hello Windows Embedded CE 6.0！"信息的消息对话框，如图9-12所示。

在这里需要特别指出的是，在这里安装的自定义模拟器在构建的过程中是需要选择很多个性化组件的。如果没有选择相应的模块，则会出现模拟器可以正常启动，但是C#程序无法执行的情况。因此需要根据

代码编写的实际需要来进行相应模块的选择。在这里建议不妨在定制模拟器的时候，选择PDA Device选项，这样就会省去Custom模式下很烦琐的选择步骤，有利于后期的程序设计。

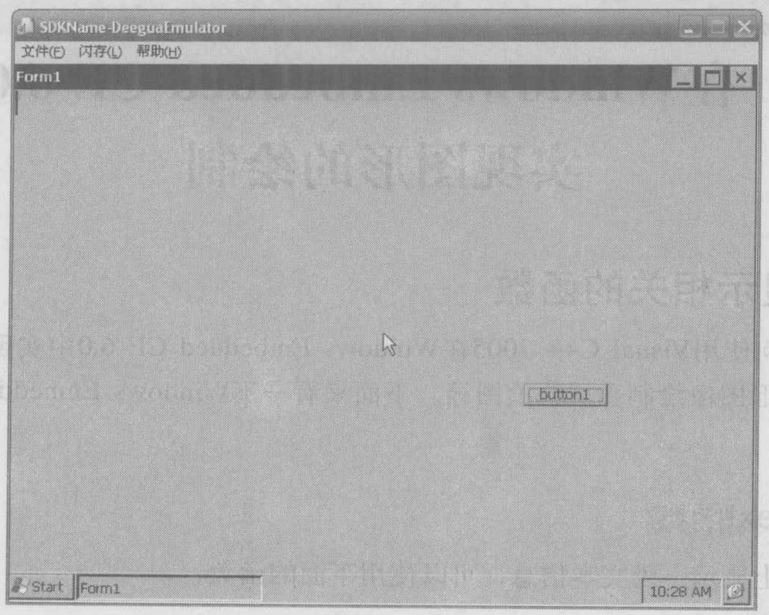


图9-11 Hello Windows Embedded CE 6.0程序的执行画面

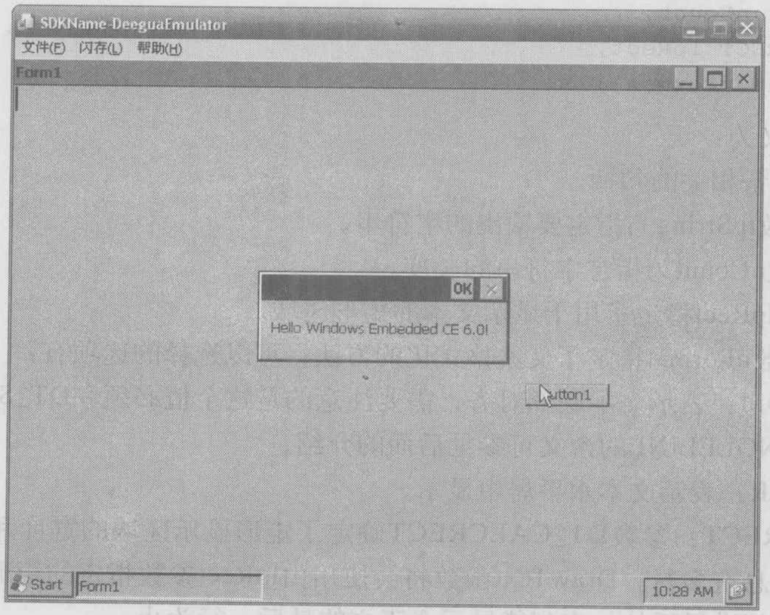


图9-12 显示的消息对话框

第10章 在Windows Embedded CE 6.0设备中实现图形的绘制

10.1 文本显示相关的函数

本节将介绍如何使用Visual C++ 2005在Windows Embedded CE 6.0中实现屏幕的绘制。主要包括文本操作、图形图像绘制和屏幕绘图等。下面来看一下Windows Embedded CE 6.0中文本操作相关的函数。

10.1.1 DrawText函数

如果想在屏幕上显示一段文本信息，可以使用下面的函数：

```
DrawText (HDC hDC,
          LPCTSTR lpString,
          int nCount,
          LPRECT lpRect,
          UNIT uFormat);
```

各个参数的含义为：

- hDC：指向设备相关的句柄。
- lpString：参数lpString为指定要输出的字符串。
- nCount：参数nCount为指定字符串的长度。
- lpRect：参数lpRect指定了用于显示文本的矩形区域。
- uFormat：参数uFormat指定了文本格式化的方法；可以选择的选项有：
  - DT_BOTTOM：表示文本底部对齐，需要注意的是这个值必须和DT_SINGLELINE同时使用。DT_SINGLELINE的含义可参见后面的介绍。
  - DT_CENTER：表示文本水平居中显示。
  - DT_CALCRECT：参数DT_CALCRECT确定了矩形显示区域的宽度和高度。如果需要显示的文本信息有多行，DrawText函数将会使用由lpRect参数指定的矩形的宽度，并且还会把矩形的底部进行延伸，直到能显示全正文的最后一行为止。
  - DT_EXPANDTABS：参数DT_EXPANDTABS表示扩展的TAB字符。
  - DT_SINGLELINE：参数DT_SINGLELINE指定了只允许单行显示文本，这个时候回车和换行均不打断原有的行显示。
  - DT_RIGHT：参数DT_RIGHT表示显示文本为右侧对齐。
  - DT_LEFT：参数DT_LEFT表示显示的文本为左侧对齐。
  - DT_TOP：参数DT_TOP表示显示的文本为顶端对齐。

- DT_VCENTER：参数DT_VCENTER表示显示的文本在垂直方向上居中对齐。
- DT_WORDBREAK：参数DT_WORDBREAK指定了单词的中断，也就是说当文本超过了参数lpRect指定的矩形的边界以后，会自动地在单词和边界之间切断正文。
- DT_TABSTOP：参数DT_TABSTOP为设置TAB字符的停止位。

10.1.2　ExtTextOut函数

也可以通过下面的函数来显示字符串：

```
BOOL ExtTextOut (HDC hdc, int X, int Y,
                 UINT fuOptions,
                 const  RECT*  lprc,
                 LPCTSTR lpString,
                 UINT cbCount,
                 const int* lpDx);
```

各个参数的含义为：

- hdc：参数hdc为设备的句柄。
- 参数X，Y为指定字符串中的字符原点的X坐标和Y坐标。
- fuOptions：参数fuOptions指定了矩形绘制区域的类型。可选项为：
  - ETO_CLIPPED：ETO_CLIPPED类型表示Windows CE系统会把字符串进行裁减，以适应矩形。
  - ETO_OPAQUE：ETO_OPAQUE类型则表示用当前背景颜色填充该矩形。
- lprc：参数lprc指向一个RECT结构。
- lpString：参数lpString指定了要显示的字符串。
- cbCount：参数cbCount指定了要显示的字符数。
- lpDx：参数lpDx如果不是零，就代表指向一个Long值数组的指针。该数组对数据对字符的间距进行了说明。其中数组中的第一个数据是第一和第二个字符的间距，第二个数据是第二和第三个字符的间距，以此类推。如果为零，函数就使用字体的默认间距设置。

10.1.3　文本的属性

在Windows Embedded CE 6.0中，文本的属性主要包括文本的背景颜色显示方式、文本颜色、文本的背景颜色等信息。下面就来看一下如何设置这些属性。

1. 文本的背景模式

在Windows Embedded CE 6.0中，文本的背景模式主要是指处理文本输出的背景颜色的显示方式。可以通过调用SetBkMode函数来实现。

函数SetBkMode()的用法为：

```
int SetBkMode (HDC hdc, int nBkMode);
```

其中参数hdc仍旧表示设备句柄，而参数nBkMode则表示本文的背景模式。它可以有下面两个选项：

- OPAQUE：参数OPAQUE表示显示文本的背景颜色。

• TRANSPARENT：参数TRANSPARENT表示使设备的背景颜色无效，即背景颜色为透明。

2. 文本的颜色

如果想在Windows Embedded CE 6.0中设置颜色，可以使用RGB函数。该函数的用法为：

```
COLORREF RGB (BYTE byRed, BYTE byGreen, BYTE byBlue);
```

其中各个参数表示相应颜色的强度，具体的取值为：

• byRed：表示红色的强度值，其取值范围为0~255。
• byGreen：表示绿色的强度值，其取值范围为0~255。
• byBlue：表示蓝色的强度值，其取值范围为0~255。

函数RGB()获得的是上述三种颜色混合后而得到的颜色，如RGB（255，0，0）表示纯红色，RGB（0，0，255）表示纯绿色等。

设置本文颜色的函数为：

```
COLORREF SetTextColor(HDC hdc, COLORREF crColor);
```

各个参数的含义为：

• hdc：指向设备的句柄。
• crColor：制定的文本的颜色。

如果想获取当前文本的颜色，则可以通过GetTextColor函数。GetTextColor函数的函数原型为：

```
COLORREF GetTextColor (HDC hdc);
```

函数中唯一的参数为指向设备的句柄。

如果想要知道当前的设备支持多少种颜色，可以使用GetDeviceCaps函数。这个函数的函数原型为：

```
int GetDeviceCaps (HDC hdc, int nIndex);
```

各个参数的含义为：

• hdc：指向设备的句柄。在这里，不同的设备有不同的性能，如打印机句柄就与显示设备的句柄不同。
• nIndex：表示被查询的设备的性能。

10.1.4 SetBkColor函数

SetBkColor函数的作用是设置背景颜色。其函数原型为：

```
COLORREF SetBkColor(HDC hdc,COLORREF crColor);
```

各个参数的含义为：

• hdc：指向设备的句柄。
• crColor：指定的背景颜色。

关于这个函数的具体用法，可以参考下面的程序实例。

10.1.5 SetBkMode函数

SetBkMode函数的作用是指定背景颜色和文字的混合方式。其函数原型为：

```
int SetBkMode (HDC hdc,int iBkMode);
```

各个参数的含义为：

- hdc：指向设备的句柄。
- iBkMode：指定了背景混合的模式。可以有下面两种选择：
  - OPAQUE：用当前背景色填充字符空隙。
  - TRANSPARENT：背景色作透明处理，也就是说不进行上面的填充。

这个函数的具体用法也可以参考下面的程序实例。

10.2　Windows Embedded CE 6.0中的字体

在Windows Embedded CE 6.0中，系统使用的字体和普通的Windows是一样的，大部分都属于比例字体，也就是说字符的宽和高是不相等的。例如，字符l和m的宽度就不一样。字符l和n的高度也不相同。同时，字符和字符之间的距离以及行与行之间的距离也是不相同的。在Windows CE中，字符的高、宽和行间距等信息都包含在字符的属性中，因此字符的属性可以较为完整地描述字符的特点。

在Windows Embedded CE 6.0中，如果想在程序中更改默认的字体，则需要通过函数首先创建一种字体。创建字体的函数为：

```
HFONT CreateFontIndirect (const LOGFONT* lplf);
```

函数中唯一的参数lplf为一个指向LOGFONT结构的指针，LOGFONT定义了字体的属性，其结构为：

```
typedef struct tagLOGFONT {LONG lfHeight;
                           LONG lfWidth;
                           LONG lfEscapement;
                           LONG lfOrientation;
                           LONG lfWeight;
                           BYTE lfItalic;
                           BYTE lfUnderline;
                           BYTE lfStrikeOut;
                           BYTE lfCharSet;
                           BYTE lfOutPrecision;
                           BYTE lfClipPrecision;
                           BYTE lfQuality;
                           BYTE lfPitchAndFamily;
                           TCHAR lfFaceName[LF_FACESIZE];
                           } LOGFONT;
```

各个参数的含义为：

- lfHeight：设置字符高度。
- lfWidth：设置字符的宽度。
- lfEscapement：文本输出的字符串走向。
- lfOrientation：每一个字符的旋转角度。
- lfWeight：字体的粗重，如表10-1所示。
- lfItalic：不为0时，表示为斜字体。

- lfUnderline：不为0时，加下划线。
- lfStrikeOut：不为0时，取消下划线。
- lfCharSet：指定产生字形的字符集。
- lfOutPrecision：指定自定义字体、字形与Windows选用的实际字型间准确程度。
- lfClipPrecision：指定字符串超过指定输出部分的切割方式。
- lfQuality：表示字符的输出质量，比如是默认质量还是草稿质量。
- lfPitchAndFamily：指定字间距与字形。
- lfFaceName：指定字形的外形名称。

表10-1 字体的粗重

磅 重	取 值	磅 重	取 值
FW_DONTCARE	0	FW_SEMIBOLD	600
FW_THIN	100	FW_DEMIBOLD	600
FW_EXTRALIGHT	200	FW_BOLD	700
FW_ULTRALIGHT	200	FW_EXTRABOLD	800
FW_LIGHT	300	FW_ULTRABOLD	800
FW_NORMAL	400	FW_HEAVY	900
FW_REGULAR	400	FW_BLACK	900
FW_MEDIUM	500		

10.3 图形绘制

在Windows Embedded CE 6.0程序中，图形绘制也是一个非常重要的部分。图形绘制主要包括直线的绘制、图形的绘制和填充等。首先来看一下直线的绘制。

10.3.1 直线的绘制

如果想在Windows Embedded CE 6.0中绘制直线，可以通过下面这个函数：

```
BOOL Polyline(HDC hdc, const POINT* lppt, int cPoints);
```

各个参数的含义为：

- hdc：参数hdc为设备句柄。
- lppt：参数lppt为一个指向POINT结构的指针。
- cPoints：参数cPoints为包含多少个POINT结构。

POINT结构原型为：

```
typedef struct tagPOINT {LONG x;
                         LONG y;
                        }POINT;
```

里面的参数X表示某个点的X坐标，参数Y表示某个点的Y坐标。如果想在Windows CE中绘制一条从（0，0）到（100，200）的点，可以通过下面的代码来实现：

```
POINT pts[2];
```

```
pts[0].x=0;
pts[0].y=0;
pts[1].x=100;
pts[1].y=200;

PolyLine(hdc,&pts,2);
```

在Windows Embedded CE 6.0中，另外一种绘制直线的方法就是用MoveToEx函数和LineTo函数。这两个函数的原型为：

```
WINGDIAPI BOOL WINAPI MoveToEx (HDC hdc, int X, int Y,
                               LPPOINT lpPoint );
```

各个参数的含义为：

- hdc：参数hdc为设备句柄。
- X：参数X为屏幕上某点的X坐标。
- Y：参数Y为屏幕上某点的Y坐标。
- lpPoint：参数lpPoint为一个指向POINT结构的指针，用来接收原来点的坐标。如果这个参数被设置成NULL，则表示不用重新标注前一个位置的坐标。

```
WINGDIAPI BOOL WINAPI LineTo(HDC hdc, int nXEnd, int nYEnd);
```

各个参数的含义为：

- hdc：参数hdc为设备的句柄。
- nXEnd：参数nXEnd为目的点的X坐标。
- nYEnd：参数nYEnd为目的点的Y坐标。

如果想使用上面两个函数绘制（0，0）到（100，200）之间的连线，可以通过下面的代码来实现：

```
MoveToEx(hdc, 0, 0, NULL);
LineTo(hdc, 100, 200);
```

这两个函数的作用在首先把坐标点移动到（0，0）位置，也就是说从某一个点移动到当前的（0，0）坐标点，然后通过LineTo函数确定终点，这样就实现了两点之间连线的绘制。

10.3.2 画笔

在Windows Embedded CE 6.0中，可以通过下面这个函数来创建一个画笔：

```
HPEN CreatePen( int fnPenStyle, int nWidth, COLORREF crColor);
```

各个参数的含义为：

- fnPenStyle：参数fnPenStyle指明了画笔的类型。它可以有下面几个选项：
  - PS_SOLID：表示画出的线为实线。
  - PS_DASH：表示画出的线为虚线。
  - PS_NULL：表示画出的线是不可见的。
- nWidth：参数nWidth表示画笔的宽度。
- crColor：参数crColor表示画笔的颜色。

在Windows Embedded CE 6.0中，如果想创建一个红颜色的实线画笔，可以通过下面的代码

来实现：

```
hPen=CreatePen (PS_SOLID, 1, RGB(0xff, 0, 0));
```

在Windows CE中，另外一个创建画笔的函数为：

```
HPEN CreatePenIndirect (const LOGPEN* lplgpn);
```

里面唯一的参数为一个指向LOGPEN结构的指针。LOGPEN结构的原型为：

```
typedef struct tagLOGPEN {UINT lopnStyle;
                          POINT lopnWidth;
                          COLORREF lopnColor;
                         }LOGPEN;
```

各个参数的含义为：

- lopnStyle：参数lopnStyle表示画笔的种类。它可以有以下几个选择：
  - PS_SOLID：实线画笔。
  - PS_DASH：虚线画笔。
  - PS_DOT：画出的线由一个个点构成。
  - PS_DASHDOT：画出的线为点划线型。
  - PS_DASHDOTDOT：画出的线为双点划线。
  - PS_NULL：画笔是不可见的。
  - PS_INSIDEFRAME：表示在一个图形设备的内部画实线。
- lopnWidth：参数lopnWidth为画笔的宽度。
- lopnColor：参数lopnColor为画笔的颜色。

如果使用CreatePenIndirect函数创建一个画笔，可以通过下面的代码来实现：

```
LOGPEN lp;
HPEN hPen;
lp.lopnStyle=PS_SOLID;
lp.lopnWidth.x=1;
lp.lopnWidth.y=1;
lp.lopnColor=RGB(0xff,0,0);
hPen= CreatePenIndirect(&lp);
```

10.3.3 刷子

在Windows Embedded CE 6.0中，刷子（brush）经常被用来填充图形，或者填充Windows窗体的背景画面。刷子实际上是一个位图，通常为8×8像素大小。

在Windows Embedded CE 6.0中创建实线类型刷子的函数为：

```
HBRUSH CreateSolidBrush (COLORREF crColor);
```

函数中的参数crColor代表了刷子的颜色。

如果想在Windows CE中创建自定义类型的刷子，则可以通过下面这个函数：

```
HBRUSH CreateDIBPatternBrushPt (const void* lpPackedDIB, UINT iUsage);
```

CreateDIBPatternBrushPt函数可以用来创建任何尺寸的刷子。在Windows Embedded CE 6.0中，

系统不支持带有阴影线的刷子。但是可以通过CreateDIBPatternBrushPt函数创建定制刷子来达到这样的效果。

10.3.4 矩形

在Windows Embedded CE 6.0中绘制一个矩形，可以通过下面这个函数来实现：

```
BOOL Rectangle (HDC hdc,
                int nLeftRect,  int nTopRect,
                int nRightRect,  int nBottomRect);
```

各个参数的含义为：

- hdc：参数hdc为设备句柄。
- nLeftRect：参数nLeftRect为起始点的X坐标。
- nTopRect：参数nTopRect为起始点的Y坐标。
- nRightRect：参数nRightRect为绘制矩形的宽度。
- nBottomRect：参数nBottomRect为绘制矩形的高度。

当使用Rectangle函数绘制矩形的时候，如果没有特殊说明，系统会用当前默认的画笔（Pen）绘制出矩形的边框，然后用当前默认的刷子（brush）进行填充。如果在Windows CE的屏幕上从（0，0）点开始，绘制一个10×20大小的矩形，可以通过下面的代码实现：

```
Rectangle(0,0,10,20)
```

再来看看下面这个函数：

```
BOOL GetClientRect (HWND hWnd,  LPRECT lpRect );
```

这个函数的作用是获得一个Windows窗体的坐标。其中参数hWnd为设备句柄，参数lpRect为存储窗体坐标的一个Rect结构。Rect结构的函数原型为：

```
typedef struct _RECT {LONG left;
                      LONG top;
                      LONG right;
                      LONG bottom;
                     }RECT;
```

各个参数的含义为：

- left：参数left表示矩形左上角的X坐标。
- top：参数top表示矩形左上角的Y坐标。
- right：参数right表示矩形右下角的X坐标。
- bottom：参数bottom表示矩形右下角的Y坐标。

当GetClientRect()函数被成功执行以后，RECT里面的top和left为0，而right和bottom则为Windows窗体的宽度和高度。

10.3.5 圆

在Windows Embedded CE 6.0中绘制圆形的函数为：

```
BOOL Ellipse (HDC hdc,
              int nLeftRect, int nTopRect,
```

```
                int nRightRect,  int nBottomRect);
```

在Windows CE中，系统是通过一个矩形来限制所绘的圆形的，因此各个参数的含义为：

- hdc：参数hdc为设备的句柄。
- nLeftRect：参数nLeftRect为外围矩形左上角的X坐标。
- nTopRect：参数nTopRect为外围矩形左上角的Y坐标。
- nRightRect：参数nRightRect为外围矩形右下角的X坐标。
- nBottomRect：参数nBottomRect为外围矩形右下角的Y坐标。

图10-1为绘制圆形的图示。

图10-1　把圆形绘制到外围矩形框内

10.3.6　圆角矩形

在Windows Embedded CE 6.0中绘制圆角矩形的函数为：

```
BOOL RoundRect (HDC hdc, int nLeftRect, int nTopRect,
                int nRightRect,  int nBottomRect,
                int nWidth,    int nHeight);
```

各个参数的含义为：

- hdc：参数hdc为设备的句柄。
- nLeftRect：参数nLeftRect为外围矩形左上角的X坐标。
- nTopRect：参数nTopRect为外围矩形左上角的Y坐标。
- nRightRect：参数nRightRect为外围矩形右下角的X坐标。
- nBottomRect：参数nBottomRect为外围矩形右下角的Y坐标。
- nWidth：参数nWidth为圆角矩形拐角处的矩形宽度。
- nHeight：参数nHeight为圆角矩形拐角处的矩形高度。

图10-2为绘制圆角矩形的图示。

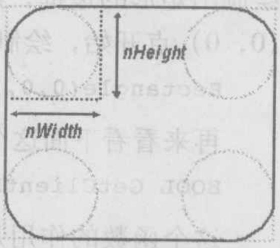


图10-2　圆角矩形的绘制图示

10.4　使用相关API函数在Windows Embedded CE 6.0中进行屏幕绘制

前面已经介绍了Windows CE中和文本显示相关的一些函数。您一定已经对这些函数中各个参数的含义有所了解，那么如何在实际的程序设计中运用这些函数呢？下面将解开这些问题的答案。

10.4.1　程序实例中涉及的函数框架

在这里，使用了DrawText和ExtTextOut两个函数进行文本的输出。同时又对文本的背景色进行了不同的设置。下面就来看一下这个程序中的一些关键代码的含义。为了使您对程序实例有一个整体上的认识，在本章的后面列出了程序实例的完整代码。

由于是第一次涉及Win32程序，因此在这里介绍一下程序框架中一些主要函数的作用，如WinMain()、InitInstance()等。下面首先来看一下WinMain函数。

程序中的WinMain函数是整个程序的入口，也就是说这个程序在运行的时候是从该函数开始的。这个函数将进行一系列的初始化工作，同时处理消息循环。其中初始化工作由InitInstance函数来完成，而下面的代码：

```
while (GetMessage (&msg, NULL, 0, 0)) {
    TranslateMessage (&msg);
    DispatchMessage (&msg);
```

就是一个消息循环语句。当激发一个消息的时候，首先调用TranslateMessage函数来检索并生成一个字符消息WM_CHAR，然后再调用DispatchMessage消息，将消息分发给相应的窗口。进行完上述操作以后，又进入到循环中，直到函数接收到WM_QUIT消息。

刚才提到InitInstance函数完成窗口的注册、初始化等工作。其中：

- style：表示窗口类的风格。
- lpfnWndProc：表示窗口过程。
- hInstance：表示该窗口类的窗口过程所属的程序实例。
- hIcon：表示窗口类所用的图标。
- hCursor：表示窗口类所用的光标。
- hbrBackground：表示窗口类所用的背景刷。
- lpszMenuNam：表示窗口类所用的菜单资源。
- lpszClassName：表示窗口类的名称。

在InitInstance函数中使用CreateWindowEx来创建一个具有扩展风格的重叠式窗口、弹出式窗口或子窗口。CreateWindowEx函数的函数原型为：

```
HWND CreateWindowEx (DWORD dwExStyle,      LPCTSTR lpClassName,
                     LPCTSTR lpWindowName, DWORD dwStyle,
                     int x,    int y,      int nWidth, int nHeight,
                     HWND hWndParent,      HMENU hMenu,
                     HINSTANCE hInstance,  LPVOID lpParam );
```

各个参数的含义为：

- dwExStyle：参数dwExStyle表示扩展窗体样式。
- lpClassName：参数lpClassName指向注册过的类名。
- lpWindowName：参数lpWindowName指向窗体名。
- dwStyle：参数dwStyle为窗体的风格。
- x：参数x为窗口的水平位置。
- y：参数y为窗口的垂直位置。
- nWidth：参数nWidth为窗口的宽度。
- nHeight：参数nHeight为窗口的高度。
- hWndParent：参数hWndParent为父窗口的句柄。
- hMenu：参数hMenu为菜单的句柄。
- hInstance：参数hInstance为应用程序的句柄。
- lpParam：参数lpParam为指向创建窗口所需的数据。

当这个函数被成功执行以后，会返回新创建窗口的句柄，否则返回NULL。

InitInstance函数中ShowWindow的作用是设置窗体的显示状态（最大化、最小化、隐藏等），UpdateWindow的作用是更新窗口。当UpdateWindow被调用的时候，会引发WM_PAINT消息，此时WM_PAINT会被直接发送到窗口过程，而不是被送到窗口队列中，因此可以立即绘制出窗口。

在程序执行中，应用程序都是通过窗口过程MainWndProc来处理消息的。它里面的4个参数的含义分别为：

- hWnd：窗口的句柄。
- wMsg：消息标示。
- wParam：消息参数1。
- lParam：消息参数2。

在这里有一点需要注意的是，应用程序中的非队列消息由Windows直接送给目的窗口的窗口过程；而队列消息则由DispatchMessage等派发给目的窗口的窗口过程。

DoPaintMain是处理窗口重绘（WM_PAINT）消息的函数，本程序实例中文本显示的函数就添加在这个函数体中。DoDestroyMain函数则为处理窗口销毁（WM_DESTROY）消息的函数。

10.4.2 程序实例中涉及的文本显示函数

既然要实现用不同的颜色来显示文字，那么就不妨把这些颜色数值存储在一个数组中，只要遍历这个数组，就可以实现对不同颜色的调用。在这个程序中，存储颜色信息的数组被命名为dwColorTable。它里面的每一个元素都是用RGB方法表示的颜色数值。在程序中的代码为：

```
DWORD dwColorTable[] = {RGB(0,0,1), RGB(0,233,1),
                        RGB(1,1,1), RGB(45,23,1),
                        RGB(254,3,5), RGB(0,254,1)};
```

设定完颜色数值以后，还要进行背景模式的设定、文字颜色的设置和背景颜色的设置。其中把背景显示模式设置成透明，这样就可以减少背景色对文字显示效果的干扰。在这里有一点需要特别提及的是，刚才提到的背景色指的是整个窗体的背景颜色，与文字显示位置的背景颜色是不同的。图10-3指明了各自所表示的范围。

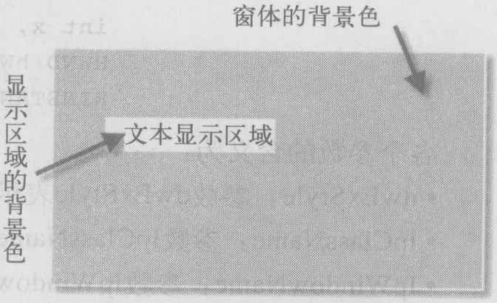


图10-3 背景颜色位置说明

在程序中，设置文字颜色使用的是SetTextColor函数。该函数在程序中的书写方式为：

```
SetTextColor (hdc, dwColorTable[i]);
```

其含义为用dwColorTable中的第i种颜色作为文字显示的颜色。很显然，里面的i为一个循环变量。设置文字背景颜色使用的是SetBkColor函数，该函数在程序中的书写方式为：

```
SetBkColor (hdc, dwColorTable[6-i]);
```

其含义为设置dwColorTable中倒数第i种颜色为文本显示区域的背景颜色。如果在这里也直接使用第i种颜色，文本颜色和背景颜色就会重叠在一起，这就无法看到文本显示的内容了。这也是在这里使用6-i的原因。

为了使读者对本章中介绍的两个文本显示函数DrawText和ExtTextOut都有所了解，在程序代码中分别使用这两种方式进行了文本信息的显示。其中DrawText在程序代码中的书写方式为：

```
DrawText (hdc, TEXT ("Using DrawText"), -1, &rect,
                  DT_VCENTER);
```

ExtTextOut在程序代码中的书写方式为：

```
ExtTextOut(hdc,rect.left,rect.top,ETO_CLIPPED|| ETO_OPAQUE,&rect,
                    TEXT("Using ExtTextOut"),16,0);
```

由于本章的前面已经对这两个函数中的参数进行了非常详细的介绍，因此在这里不再赘述二者所表示的含义了。不清楚的读者可参考本章前面相关内容的介绍。

为了使得各个显示信息都可以很清晰地显示出来，在这里使用了一个循环语句来实现显示位置坐标的递增变化。

至此，该程序实例的关键代码都已经介绍完毕。为了体现实验的动手操作能力，建议在看懂关键函数的基础上，亲自动手进行一下相关函数的修改。例如，更改一下ExtTextOut函数中的参数，更改一下标题信息等，看看实际的显示效果和预计的是否一样。

在这里有一点需要特别说明的是，由于Windows Embedded CE 6.0模拟器是英文版本的，不支持中文显示，因此建议读者不要把显示信息更改成中文，否则会显示乱码。

10.4.3 程序实例中涉及的图形绘制函数

在这个程序实例中，用到的图形绘制函数有：

- 圆角矩形绘制函数RoundRect
- 圆形绘制函数Ellipse
- 矩形绘制函数Rectangle
- 直线绘制函数Polyline

下面就来分别看一下这些函数在程序实例中的具体应用。

圆角矩形绘制函数RoundRect在程序实例中的代码为：

```
RoundRect (hdc, 300, 50, 500, 100, 50, 50);
```

其含义为绘制一个左上角坐标为（300，50），右下角坐标为（500，100），拐角处矩形的宽度为50，高度为50的圆角矩形。感兴趣的读者可以尝试把RoundRect函数中的最后两个参数进行一下修改，如修改成大小不等的两个数值，看看效果会有什么不同。该函数的运行效果如图10-4所示。

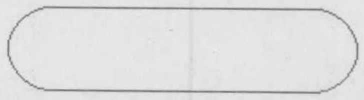


图10-4 RoundRect函数执行效果

圆形绘制函数Ellipse在程序实例中的代码为：

```
Ellipse (hdc, 300, 110, 500, 200);
```

其含义为绘制一个外接矩形的左上角坐标为（300，110），右下角坐标为（500，200）的椭圆。实际上这个函数之所以绘制出一个椭圆是因为外接矩形的宽和高是不相等的。具体数据为：

外接矩形的宽度：500－300=200。

外接矩形的高度：200－110=90。

当把这个差值修改成相当大小的时候，就会绘制出一个圆形。

图10-5展示了Ellipse函数的运行效果。

图10-5 Ellipse函数的运行效果

矩形绘制函数Rectangle在程序实例中的代码为：

```
Rectangle (hdc, 300, 250, 500, 500);
```

其含义为绘制一个左上角坐标为（300，250），右下角坐标为（500，500）的矩形。图10-6展示了Rectangle函数的运行效果。

直线绘制函数Polyline在程序实例中的代码为：

```
POINT pt[3];
pt[0].x=100;
pt[0].y=200;
pt[1].x=200;
pt[1].y=300;
pt[2].x=200;
pt[2].y=500;
Polyline(hdc,pt,3);
```

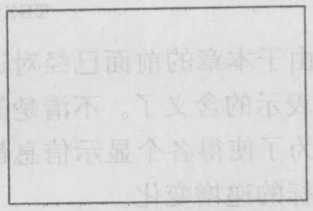


图10-6 Rectangle函数的运行效果

其含义为绘制一个从（100，200）到（200，300），再到（200，500）的直线。图10-7展示了Polyline函数的运行效果。

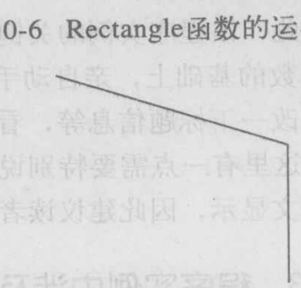


图10-7 Polyline函数的运行效果

10.4.4 程序实例运行效果

图10-8展示了程序的运行效果。

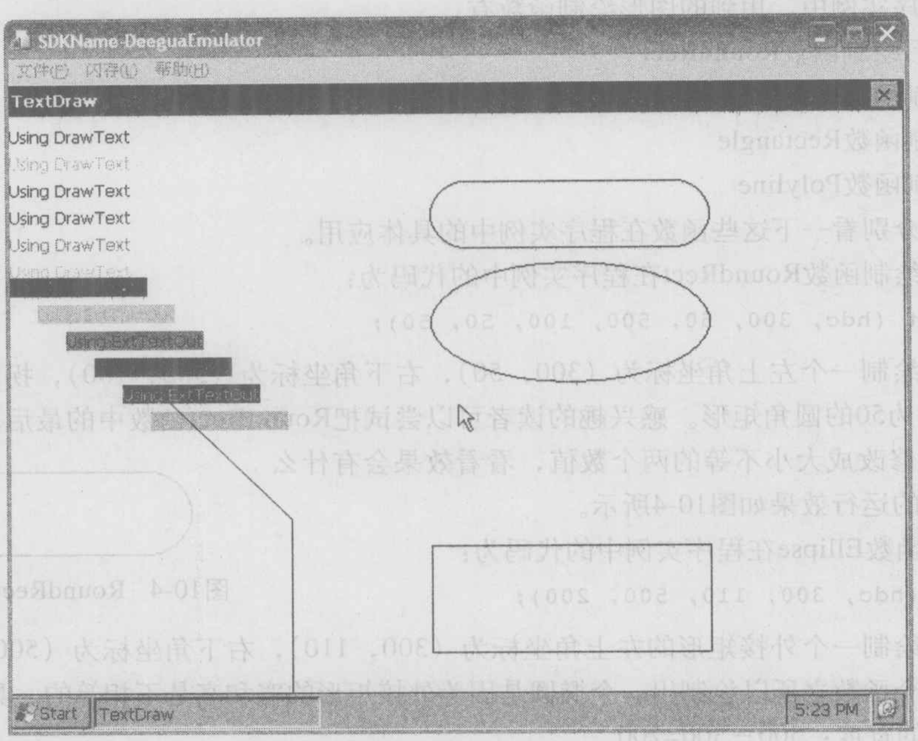


图10-8 程序运行效果

10.4.5 程序源代码——TextDraw.h

```
#pragma once
#ifdef STANDARDSHELL_UI_MODEL
#include "resource.h"
#endif
#ifdef POCKETPC2003_UI_MODEL
#include "resourceppc.h"
```

```
#endif

#define dim(x) (sizeof(x) / sizeof(x[0]))

struct decodeUINT {
    UINT Code;

    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {
    UINT Code;
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);
};

HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

LRESULT DoPaintMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);
```

10.4.6　程序源代码——TextDraw.cpp

```
// TextDraw.cpp : Defines the entry point for the application.
//

#include "stdafx.h"
#include "TextDraw.h"
#include <windows.h>
#include <commctrl.h>

const TCHAR szAppName[] = TEXT ("TextDraw");
HINSTANCE hInst;                        // Program instance handle

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_PAINT, DoPaintMain,
    WM_DESTROY, DoDestroyMain,
};
//======================================================================
// Program Entry Point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    int rc = 0;
```

```
    HWND hwndMain;

    // Initialize this instance.
    hwndMain = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;

     // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//----------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow){
    WNDCLASS wc;
    HWND hWnd;

    hInst = hInstance;    // Save handle in global variable.

#if defined(WIN32_PLATFORM_PSPC) || defined(WIN32_PLATFORM_WFSP)
    // If Windows Mobile, allow only one instance of the application.
    hWnd = FindWindow (szAppName, NULL);
    if (hWnd) {
        SetForegroundWindow ((HWND)(((DWORD)hWnd) | 0x01));
        return 0;
    }
#endif
    // Register application main window class.
    wc.style = 0;                              // Window style
    wc.lpfnWndProc = MainWndProc;              // Callback function
    wc.cbClsExtra = 0;                         // Extra class data
    wc.cbWndExtra = 0;                         // Extra window data
    wc.hInstance = hInstance;                  // Owner handle
    wc.hIcon = NULL,                           // Application icon
    wc.hCursor = LoadCursor (NULL, IDC_ARROW);// Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                   // Menu name
    wc.lpszClassName = szAppName;              // Window class name

    if (RegisterClass (&wc) == 0) return 0;

    // Create main window.
```

```
    hWnd = CreateWindowEx (WS_EX_NODRAG,          // Ex Style flags
                          szAppName,              // Window class
                          TEXT("TextDraw"),       // Window title
                          // Style flags
                          WS_VISIBLE | WS_CAPTION | WS_SYSMENU,
                          CW_USEDEFAULT,          // x position
                          CW_USEDEFAULT,          // y position
                          CW_USEDEFAULT,          // Initial width
                          CW_USEDEFAULT,          // Initial height
                          NULL,                   // Parent
                          NULL,                   // Menu, must be null
                          hInstance,              // Application instance
                          NULL);                  // Pointer to create
                                                  // Parameters
    // Return fail code if window not created.
    if ((!hWnd) || (!IsWindow (hWnd))) return 0;

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
//----------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {
    return nDefRC;
}
//======================================================================
// Message handling procedures for MainWindow
//
//----------------------------------------------------------------------
// MainWndProc - Callback function for application window
//
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
```

```
//----------------------------------------------------------------------
// DoPaintMain - Process WM_PAINT message for window.
//
LRESULT DoPaintMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                     LPARAM lParam) {
    PAINTSTRUCT ps;
    RECT rect;
    HDC hdc;
    INT i, dy;
    DWORD dwColorTable[] = {RGB(0,0,1), RGB(0,233,1),
                            RGB(1,1,1), RGB(45,23,1),
                            RGB(254,3,5), RGB(0,254,1)};

    GetClientRect (hWnd, &rect);

    hdc = BeginPaint (hWnd, &ps);

    dy = 20;

    rect.bottom =30;
    SetBkMode (hdc, TRANSPARENT);
    for (i = 0; i < 6; i++) {
        SetTextColor (hdc, dwColorTable[i]);
        SetBkColor (hdc, dwColorTable[6-i]);

        DrawText (hdc, TEXT ("Using DrawText"), -1, &rect,
                  DT_VCENTER );
        rect.top += dy;
        rect.bottom += dy;

    }

    SetBkMode (hdc, OPAQUE);

    for (i = 0; i < 6; i++) {
        SetTextColor (hdc, dwColorTable[i]);
        SetBkColor (hdc, dwColorTable[6-i]);

        ExtTextOut(hdc,rect.left,rect.top,ETO_CLIPPED|| ETO_OPAQUE,&rect,
                   TEXT("Using ExtTextOut"),16,0);

        rect.top += dy;
        rect.left += dy;
    }
```

```
    hdc = BeginPaint (hWnd, &ps);

    RoundRect (hdc, 300, 50, 500, 100, 50, 50);

    Ellipse (hdc, 300, 110, 500, 200);

    Rectangle (hdc, 300, 320, 500, 400);

    POINT pt[3];
    pt[0].x=100;
    pt[0].y=200;
    pt[1].x=200;
    pt[1].y=300;
    pt[2].x=200;
    pt[2].y=500;
    Polyline(hdc,pt,3);

    EndPaint (hWnd, &ps);
    return 0;

}
//----------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
```

第11章 Windows Embedded CE 6.0中的输入处理

在经常使用的Windows系统中，通常的输入方式为键盘和鼠标。在Windows Embedded CE 6.0中，仍旧支持这两种输入方式。但是由于嵌入式系统的一些特点，在Windows Embedded CE系统中更多是通过触摸屏幕的方式来实现和系统的交互。本章将介绍这方面的相关内容。

11.1 输入处理概述

在一个嵌入式系统中，实现对系统信息的输入是用户与系统通信的方式之一，同时针对不同的嵌入式系统，其输入方式也不尽相同，如银行自动取款机，可以通过按键的方式进行输入，也可以通过屏幕点击的方式输入。而在本书介绍的Windows Embedded CE 6.0设备中，系统主要支持鼠标、键盘、触摸屏、软键盘、手写输入等。

在Windows Embedded CE 6.0中，设备的输入管理是由一套子系统来完成的。这个子系统负责处理前台窗口、活动窗口和焦点窗口。每一个线程有一个特定的窗口，被称为活动窗口。这是被特定线程拥有和激活的最高等级的窗口；而焦点窗口则是可以接收来自键盘的消息的窗口。

前台窗口、活动窗口和焦点窗口是相互关联的，通过窗口的改变可以改变输入焦点，通过焦点的改变也可以改变活动窗口。活动窗口和焦点窗口的信息都被保存在消息队列中，所以它们都是一个基本的线程。当调用SetActiveWindow时，一个线程将把和它同一个线程的窗口激活。函数SetFocus将会改变其所在线程中任意窗口的输入焦点。当焦点被改变或者别的窗口被激活的时候，WM_SETFORCUS、WM_KILLFOCUS将会被发送。

在Windows Embedded CE 6.0系统中，如果得到的是一个键盘输入事件，系统的事件管理器将会将与其信息相关的窗口和消息队列交给前台线程处理后返回。相应的信息会表现为函数PostMessage所发送的消息。而如果得到的事件是一个触摸屏输入事件，系统的事件管理器会调用窗口管理器来找出是哪一个窗口被击中，接着使用PostMessage函数将消息打包，然后放到所属窗口合适的消息队列中。

11.2 键盘输入方式

在Windows Embedded CE 6.0系统中，尽管可以使用触摸屏代替键盘来进行一系列的操作，但是在输入大量数据的时候，仍旧需要采用键盘的方式。有的Windows Embedded CE系统，如Pocket PC或者Pocket PC Phone Edition等没有键盘，但是在系统中可以采用软键盘的方式，通过触摸屏来模拟键盘的输入。需要特别指出的是，在Windows CE系统中，从外接键盘和软键盘进行输入是没有区别的。图11-1展示了Windows Embedded CE设备中的软

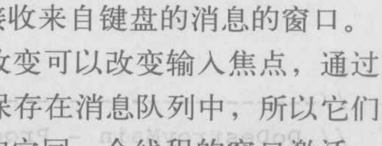


图11-1 Windows Embedded CE 6.0中的软键盘

键盘。

当按下键盘上一个按键或者松开一个按键的时候，将会产生一个扫描码。然后，键盘的驱动程序会将每一个扫描码翻译或者映射成虚键码。因为键盘的布局由于语言的不同而不同，因此Windows CE只提供在所有键盘中都能够找到的核心虚拟码。这个虚拟码集包括拉丁字母、数字和一些重要的按键（如多功能键和方向键等）。图11-2为Windows Embedded CE中中文软键盘的布局，图11-3为Windows Embedded CE中英文软键盘的布局。

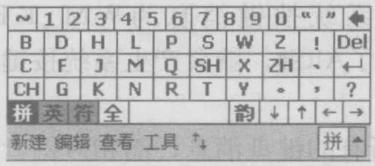


图11-2 Pocket PC中的拼音软键盘

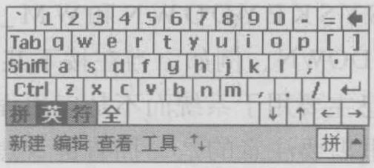


图11-3 Pocket PC中的英文软键盘

除了映射以外，键盘驱动程序还需要确定虚键究竟是产生哪一个字符。因为同一个虚键在其他按键（如Shift等）的组合状态下可以产生不同的字符。Windows CE系统在将扫描码翻译为虚键码之后，设备驱动程序会创建一条包括扫描码、虚键码和字符数据的键盘消息，然后将这条键盘消息放在系统的消息队列中，最后系统将它邮寄给适当线程的消息队列，同时删除系统消息队列中的消息。

11.2.1 输入焦点

在Windows Embedded CE中，一个时刻只能有一个窗口获得系统的输入焦点。例如，具有输入焦点的窗口的标题栏是蓝色的，否则就是灰色的。如果一个文本框具有输入焦点，就会有光标在里面闪烁，该文本框就可以立即接受到用户的键盘输入。同样，如果一个按钮具有输入焦点，按下回车键就相当于按下了该按钮。

在Windows Embedded CE 6.0系统中，可以通过GetFocus()函数来获得焦点窗口的句柄。这个函数的函数原型为：

```
HWND GetFocus (void);
```

该函数如果被成功执行，将会返回获得输入焦点的窗口的句柄。

如果想使某个窗口获得输入焦点，则可以通过SetFocus()函数来实现。SetFocus函数的函数原型为：

```
HWND SetFocus (HWND hWnd);
```

这个函数中唯一参数表示将要获得焦点的窗口的句柄。在程序设计的过程中，可以通过SetFocus()函数来实现在窗口中焦点的移动，同时当焦点发生变化的时候，系统还会向失去焦点的窗口发送WM_KILLFOCUS消息，向获得焦点的窗口发送WM_SETFOCUS消息。

11.2.2 系统的键盘消息

在Windows Embedded CE 6.0系统中，窗口以按键消息和字符消息的形式接收键盘的输入。其中，按键消息用来控制窗口的特征，字符消息用来在窗口中显示相应的文本信息。下面就来具体地看一下。

1. 键盘输入的处理过程

当按下某个按键的时候，Windows Embedded CE系统会产生一个WM_KEYDOWN或者WM_SYSKEYDOWN消息。如果长时间按下某个按键，系统会产生重复的WM_KEYDOWN或者WM_SYSKEYDOWN消息；当松开某个按键的时候，系统会产生WM_KEYUP或者WM_SYSKEYUP消息。

同时，在Windows CE中，系统可以区分出系统按键消息和非系统按键消息。系统按键消息为WM_SYSKEYDOWN和WM_SYSKEYUP；非系统按键消息为WM_KEYUP和WM_KEYDOWN。例如，当在按下某个按键的同时，按下了Alt键，就会产生系统按键消息。系统按键消息主要用于系统而不是应用程序上。

在这里需要注意的是，拥有键盘焦点的窗口可以接收所有的键盘消息，而相应键盘输入的应用程序通常只会处理WM_KEYDOWN消息。在Windows CE中，当系统中的窗口接收到WM_KEYDOWN消息以后，要检查消息中包含的虚键码，以便确定如何处理按键。虚键码通常包含在消息的wParam参数中。

既然在Windows CE系统中，按键消息分为系统按键消息和非系统按键消息，那就表示如果想输入一个字符，Windows CE系统并不会自动地产生字符消息。而是首先产生按键消息，然后再将按键消息转换成相应的字符消息。如果要想实现这个转换，需要调用TranslateMessage函数。这个函数将会检查按键消息的虚键码，如果虚键码直接对应于一个字符，就会将一条字符消息放在消息队列中，此消息将会在下一次消息循环中被分发或者被删除。在Windows Embedded CE 6.0系统中，包括4条字符消息。它们分别为WM_CHAR、WM_SYSCHAR、WM_DEADCHAR和WM_SYSDEADCHAR。

当TranslateMessage函数转换WM_KEYDOWN消息的时候，WM_CHAR字符消息将会被发送；当TranslateMessage函数转换WM_SYSKEYDOWN消息的时候，WM_SYSCHAR字符消息将会被发送；当TranslateMessage函数转换WM_KEYDOWN消息的时候，WM_DEADCHAR字符消息将会被发送；当TranslateMessage函数转换WM_SYSKEYDOWN消息的时候，WM_SYSDEADCHAR字符消息将会被发送。

不过需要注意的是，只有在同时按下ALT按键的时候，WM_DEADCHAR和WM_SYSDEADCHAR字符消息才会被向系统传递。在Windows CE系统中，典型的窗口过程将会忽略除WM_CHAR之外的所有字符消息。因为WM_CHAR消息包含了字符码和有关字符的附加数据的标志。同时，当窗口过程接收到WM_CHAR消息的时候，应该检查消息中包含的字符码，以便确定如何处理字符，字符码包含在消息的wParam参数中。

我们一定会注意到，在系统中可以接收键盘按键输入的窗口除了将会显示用户的输入以外，还应该有一个闪烁的光标来指示下一个字符将要输入的位置。因此，窗口应该在获得焦点的同时，创建并显示提示符；而在失去焦点的时候，隐藏并删除提示符。在Windows CE系统中，这些操作是在系统处理WM_SETFOCUS和WM_KILLFOCUS时执行的。可以通过CreateCaret、ShowCaret、DestroyCaret和HideCaret函数来控制光标的可见性；通过SetCaretPos函数改变光标的位置。

2. 与光标处理相关的函数

其中CreateCaret函数的作用是创建一个光标，其函数原型为：

```
BOOL CreateCaret (HWND hWnd, HBITMAP hBitmap,
                  int nWidth, int nHeight );
```

各个参数的含义为：

• hWnd：参数hWnd为指向窗口的句柄。

• hBitmap：参数hBitmap在Windows CE中应该被设置成NULL。

• nWidth：参数nWidth为光标的宽度。

• nHeight：参数nHeight为光标的高度。

在创建完一个光标以后，接下来就需要在屏幕上显示出来，显示光标的函数为ShowCaret，其函数原型为：

```
BOOL ShowCaret (hWND);
```

其中参数hWnd表示要插入光标的窗口的句柄，当函数被成功执行以后，将会返回一个非零值。

如果想释放一个光标，则可以采用下面的函数：

```
BOOL DestroyCaret (void);
```

当函数被成功执行以后，将会返回一个非零值。

如果只想在屏幕上隐藏光标而不释放它，可以通过下面的函数：

```
BOOL HideCaret (HWND hWnd);
```

3. 按键的状态

首先来看一下下面这个函数：

```
SHORT GetKeyState (int nVirtKey);
```

这个函数的作用是判断Shift、Ctrl、Alt等按键是否处于触发状态，也就是说这些按键是否处于按下或者抬起状态。我们知道，我们使用的键盘通常都有两个Shift按键、两个Ctrl按键和两个Alt按键。通过GetKeyState函数仍然可以得知到底按下的是键盘左侧的Shift，还是键盘右侧的Shift。

这个函数中的参数nVirtKey表示定义的虚拟键。若要求的虚拟键是字母或数字（A～Z，a～z或0～9），nVirtKey必须被置为相应字符的ASCII码值。该函数的返回值为给定虚拟按键的状态。若高位为1，则按键处于DOWN状态，否则为UP状态；若低位为1，则按键被触发，如CAPS LOCK键；若低位置为0，则按键被关闭，且不被触发。触发键在键盘上的表现为指示灯的亮灭。当按键被触发时灯亮，按键不被触发时灯灭。

在Windows CE中，如果想判断函数被调用时指定虚拟键的状态，则可以通过下面这个函数：

```
SHORT GetAsyncKeyState (int vKey);
```

里面的参数为256个虚拟键码之一。

4. 模拟键盘操作的函数

如果在Windows Embedded CE 6.0中，想用程序模拟键盘操作，可以使用下面这个函数：

```
VOID keybd_event (BYTE bVk, BYTE bScan,
                  DWORD dwFlags, DWORD dwExtraInfo );
```

各个参数的含义为：

• bVk：参数bVk表示虚拟键值。

- bScan：参数bScan为按键的扫描码，一般情况用0代替。
- dwFlags：参数dwFlags为选项标志。如果选项为KEYEVENTF_KEYUP，则表示模拟Keyup事件；如果选项为KEYEVENTF_SILENT，则表示按下一个按键，但是在这个按键事件中，不会模拟出按键被按下的声音。
- dwExtraInfo：参数dwExtraInfo一般被设置成0。

在使用keybd_event函数的时候，有一点需要注意，如果想要完整地模拟一个按键事件，一般需要调用两次keybd_event函数。其中一次不用KEYEVENTF_KEYUP参数，用来模拟按键被按下；另外一次使用KEYEVENTF_KEYUP参数，用来模拟按键的释放。

由于一般的键盘上通常都有两个Shift按键或者两个Ctrl按键，因此使用虚拟键码的时候需要明确地指出是左边的Shift还是右边的Shift，通常采用的虚拟键码为VK_LSHIFT和VK_RSHIFT。

5. 其他相关函数

在Windows CE中，其特有的一个函数是：

```
BOOL PostKeybdMessage (HWND hwnd,  UINT VKey,
                       KEY_STATE_FLAGS KeyStateFlags,
                       UINT cCharacters,
                       UINT* pShiftStateBuffer,
                       UINT* pCharacterBuffer );
```

这个函数的作用是送一系列的键到指定的窗口。其各个参数的含义为：

- hwnd：参数hwnd为目标窗口的句柄。
- VKey：参数VKey在Windows CE中被设置成0。
- KeyStateFlags：参数KeyStateFlags表示按键的状态。
- cCharacters：参数cCharacters指定了要发送键的数量。
- pShiftStateBuffer：参数pShiftStateBuffer指向一个数组，包含了每个键的Shift状态。
- pCharacterBuffer：参数pCharacterBuffer指向了一个缓冲区，里面包含了虚拟键码。需要注意的是，这个参数不能被设置成NULL。

最后一个键盘函数是MapVirtualKey。这个函数的作用是把虚拟键码翻译成字符。尽管在Windows的其他版本里MapVirtualKey函数会进行键盘扫描码和虚拟码之间的转换，但是在Windows CE中，系统不会这样做。MapVirtualKey函数的函数原型为：

```
UINT MapVirtualKey (UINT uCode, UINT uMapType);
```

各个参数的含义为：

- uCode：参数uCode表示待翻译的虚拟键码。
- uMapType：参数uMapType指出如何翻译该键码。

在使用这个函数的时候需要注意，MapVirtualKey函数依赖于键盘设备的驱动程序。但是有许多OEM设备并不实现这种支持，因此在这种系统上，调用MapVirtualKey函数会失败。

11.2.3 键盘测试

在Windows Embedded CE 6.0系统中，如果想要判断系统里是否有键盘，可以调用GetKeyboardStatus函数。GetKeyboardStatus函数的函数原型为：

```
DWORD GetKeyboardStatus (VOID);
```

该函数的返回值可以有下面两种情况：

- 如果系统里有键盘，则该函数返回KBDI_KEYBOARD_PRESENT标志；
- 如果键盘启用，则该函数返回KBDI_KEYBOARD_ENABLED标志。

如果想要系统中的键盘失效，则可以使用EnableHardwareKeyboard函数。该函数的函数原型为：

```
BOOL EnableHardwareKeyboard( BOOL bEnable);
```

可以通过把bEnable设置为FALSE来使系统中的键盘失效。那么什么时候需要设置键盘失效呢？如在类似平板电脑的设备中，当把屏幕扣在键盘上，用户使用手写笔进行手写输入的时候，禁止掉系统键盘就可以防止由于用户对键盘的意外按压导致的错误操作的发生。

11.3 系统中鼠标的输入

作为一款嵌入式操作系统，Windows Embedded CE和经常使用的桌面操作系统有很大的不同。在实际的应用中，Windows CE系统很少会通过鼠标来进行操作。大多情况下是用手写笔、触摸屏和软键盘来进行输入。但是为了提高Windows CE系统的可扩展性，对于添加了鼠标的Windows CE系统来说，其编程接口和桌面操作系统是一样的。

对于Windows CE系统中的应用程序而言，手写笔和触摸屏运行时候产生的事件与鼠标产生的事件基本上是一样的。更为确切地说，这些事件应该是鼠标事件的一个子集。当用户在屏幕上按下或者松开手写笔的时候，Windows CE系统中的应用程序会像处理单击鼠标左键一样来处理这些事件。当用户在屏幕上通过手写笔进行光标移动的时候，Windows CE的应用程序也会像处理鼠标的移动事件一样来处理它们。

在Windows Embedded CE系统中，无论鼠标在什么时候移动，光标下的最顶层窗口都会收到一个WM_MOUSEMOVE消息；当用户单击鼠标左键或者右键的时候，窗口会收到WM_LBUTTONDOWN或WM_RBUTTONDOWN消息；当用户按下并释放鼠标滚轮的时候，窗口会收到WM_MBUTTONDOWN及WM_MBUTTONUP消息。

对所有的这些消息，wParam和IParam都具有相同的值。wParam包含一个标志位集合，用来指出当前键盘上Ctrl或Shift键是否被按下。同其他的Windows版本一样，在这些消息里没有提供Alt键的状态。要想获得消息发送时Alt键的状态，可以使用GetKeyState函数。IParam包含两个16位的值，用来指出单击点在屏幕上的位置。低16位是相对窗口客户区左上角的X坐标位置，高16位是Y坐标位置。

当用户在屏幕的某一个位置双击鼠标的时候，Windows会向被双击的窗口发送WM_LBUTTONDBLCLK消息。不过在这里有一点需要注意，只有当窗口类注册了CS_DBLCLKS风格时系统才会发送该消息。

既然不同的鼠标操作会导致不同的鼠标消息，那么就可以通过对比发送给窗口的消息来区分单击和双击事件。当双击鼠标按键的时候，窗口首先收到来自最初单击的WM_LBUTTONDOWN和WM_LBUTTONUP消息。接下来，WM_LBUTTONDBLCLK消息会在WM_LBUTTONUP后发出。

当用户滚动鼠标滚轮的时候，窗口会收到WM_MOUSEWHEEL消息。对于该消息，IParam的内容和其他鼠标消息的内容一样，都是鼠标光标的水平和垂直位置。wParam的低字位也是同样

的标志位，指出当前被按下的键。但wParam的高字位则包含的是滚轮的距离，用常量WHEEL_DELTA的倍数来表示滚动的距离。如果该值为正，表示滚轮是朝远离用户方向滚动；如果该值为负，表示滚轮是朝用户方向滚动。

11.4 Windows Embedded CE 6.0中的触摸屏

在Windows Embedded CE 6.0系统中经常使用的触摸屏和手写笔这一组合对于传统的Windows平台使用者来说还是比较陌生的。可以把Windows CE中的手写笔看成是一个单键的鼠标。手写笔产生的鼠标消息，同其他版本的Windows以及有鼠标的Windows CE里的鼠标消息是相同的，只不过鼠标和手写笔所表现的物理实体是不同的。

在这里有一点需要注意，如果使用手写笔来进行输入，是不需要光标指示当前位置的。只有当手写笔和屏幕接触的时候，触摸屏才可以检测到手写笔的落点位置。

如果在Windows CE系统中使用手写笔进行输入，由于没有鼠标光标，因此应用程序不能通过改变光标的外形来为用户提供反馈信息。为了解决这个问题，基于触摸屏的Windows Embedded CE系统为这种典型的Windows反馈方式提供了光标设置功能。可以通过SetCursor函数来显示繁忙状态下的沙漏光标图案，这和其他版本的Windows里的应用程序使用的方式一样。

11.4.1 手写笔消息

当在Windows Embedded CE系统中使用手写笔在屏幕上操作的时候，如果手写笔与屏幕接触点的下方窗口此前没有输入焦点，此时会获得输入焦点。然后该窗口会接收到WM_LBUTTONDOWN消息；当用户抬起手写笔时，窗口会收到WM_LBUTTONUP消息；当手写笔在按下的同时，还在同一个窗口内移动，窗口就会接收到WM_MOUSEMOVE消息。

11.4.2 电子墨水

在前面的介绍中已经提到过，在Windows Embedded CE系统中用户可以通过手写笔来输入信息。因此，对手持设备来说最典型的应用就是把屏幕上用户的笔迹记录并存储下来。在这里提到的笔迹保存和手写识别的概念是不一样的。这个过程不需要对手写字进行识别，只需要把用户的“墨迹”保存下来即可。在开始阶段，完成这个功能的最好方法应该是存储由WM_MOUSEMOVE消息传入的手写笔的各个点。但是如果这样做，就会带来一些问题。如在一些性能相对较差的小型设备上，当用户的手写速度很快，而Windows CE设备却无法快速地发送消息的时候，就会导致墨迹的不连贯，以至于存储的信息无法得到满意的分辨率。针对这种问题，在Windows Embedded CE下，系统增加了一个函数来帮助程序员追踪用户的手写笔迹。这个函数为：

```
BOOL GetMouseMovePoints (PPOINT pptBuf,
                         UINT nBufPoints,
                         UINT *pnPointsRetrieved);
```

各个参数的含义为：

- pptBuf：参数pptBuf为指向POINT结构缓冲区的一个指针。这个缓冲区里面记录了手写笔移动过程中所经过的点。
- nBufPoints：参数nBufPoints为缓冲区的大小。

• pnPointsRetrieved：参数pnPointsRetrieved为一个指向整数的指针，用来接收返回给应用程序的墨迹点的点数。

当GetMouseMovePoints函数一旦接收完，这些附加的点可以用来填充上一个WM_MOUSEMOVE消息和当前WM_MOUSEMOVE消息之间的空白。在这里需要的注意的是，GetMouseMovePoints函数并不在屏幕上绘制点，只是把所有点的信息存储起来。这样做的目的是较容易地收集墨迹点，然后用相应的函数把这些点绘制出来。

11.4.3 关于输入焦点

在Windows Embedded CE中，由于涉及手写笔操作，因此在这里简单介绍一下手写笔操作过程中输入焦点发生的一些变化。在本章的前面已经提到，当用户使用手写笔按压到一个窗口的时候，系统的输入焦点将会发生变化。但是当使用手写笔从一个窗口拖动到另一个窗口的时候，新的窗口并不会获得输入焦点。因为只有手写笔在按压下以后才会对窗口设置焦点，而单纯地拖动手写笔是不会发生作用的。

当使用的手写笔移动出窗口的时候，该窗口就会停止接收WM_MOUSEMOVE消息，但是却会继续保持输入焦点。这是因为手写笔的笔尖依然下压，所以没有其他窗口会接收WM_MOUSEMOVE消息。关于这一点，与在传统的PC上面使用鼠标按键按下，并拖动出一个窗口很类似。

在Windows Embedded CE中，如果想在手写笔已经移动到窗口外的时候仍旧可以继续接收到手写笔的消息，只需要使用程序函数SetCapture即可。这个函数的函数原型为：

```
HWND SetCapture (HWND hWnd);
```

函数中唯一的参数表示接收鼠标消息的窗口的句柄。该函数会返回前一个捕捉鼠标消息的窗口的句炳，如果在这之前没有捕捉过，则会返回NULL。如果想停止接收手写笔输入产生的鼠标消息，则可以使用ReleaseCapture函数。它的函数原型为：

```
HWND ReleaseCapture (HWND hWnd);
```

函数中的参数同样为需要捕捉鼠标消息的窗口的句柄。或许会有这样的疑问，当使用了上面这个函数以后，是否一定可以保证手写笔信息被捕捉了呢？是否有什么函数可以查看呢？这个函数就是GetCapture。它的函数原型为：

```
HWND GetCapture(void);
```

这个函数的作用是返回捕捉手写笔的窗口的句柄。如果没有窗口捕捉到手写笔的输入，则返回0。在这里有一点需要注意的是，该函数的返回值为0只表示该线程没有捕捉到鼠标，并不表示其他线程或进程没有捕捉到鼠标。也就是说，捕捉手写笔的窗口必须和调用该函数的窗口在同样的线程环境里。这就意味着，如果手写笔被另一个窗口捕捉了，GetCapture依然返回0。如果一个窗口捕捉了手写笔而另一个窗口调用了GetCapture，那么最初捕捉手写笔的窗口会收到一个WM_CAPTURECHANGED消息。消息的lParam参数中包含了获得手写笔捕捉窗口的句柄。

在这里有一点需要注意的是，通常情况下，Windows CE中的手写笔是共享资源。因此编写的应用程序在使用手写笔消息一段时间以后，应该能够很好地处理手写笔丢失的情况。

11.4.4 右键单击

在经常使用的Windows系统中，在单击鼠标右键的时候，通常会调出上下文相关的、独立的

菜单。菜单里会显示出和具体功能相关的集合。此时，Windows系统会发送WM_RBUTTONDOWN和WM_RBUTTONUP消息，以表明有右键被单击了。但是当使用手写笔的时候，一般情况下是不带右键的。针对这种情况，在Windows CE中，系统允许通过手写笔来模拟右键的单击。具体的做法为，当按下Alt按键的同时用书写笔单击屏幕的时候，程序就会默认为鼠标右键被单击，此时就会显示出上下文相关的菜单。

在Windows Embedded CE 6.0系统中，要想实现按压Alt键的同时单击屏幕实现模拟鼠标右键，可以通过SHRecognizeGesture函数来实现。SHRecognizeGesture函数的函数原型为：

```
WINSHELLAPI DWORD SHRecognizeGesture(SHRGINFO * shrg);
```

这个函数中的参数shrg为一个指向SHRGINFO结构的指针。SHRGINFO结构的定义形式如下：

```
typedef struct tagSHRGI {DWORD cbSize;
                         HWND hwndClient;
                         POINT ptDown;
                         DWORD dwFlags;
                        }SHRGINFO, *PSHRGINFO;
```

各个参数的含义为：

- cbSize：参数cbSize表示SHRGINFO结构的大小。
- hwndClient：参数hwndClient需要设置为调用该函数的窗口的句柄。
- ptDown：参数ptDown为鼠标落点的坐标。
- dwFlags：参数dwFlags表示相应的标志；它可以有下面的选择：
  - SHRG_RETURNCMD：SHRG_RETURNCMD参数表示如果用户已经下压手写笔，并且保持按压姿势，函数就会返回GN_CONTEXTMENU；否则会返回0。
  - SHRG_NOTIFYPARENT：SHRG_NOTIFYPARENT参数表示如果系统识别出用户正确操作，就给父窗口发送一个WM_NOTIFY消息。
  - SHRG_LONDELAY：SHRG_LONDELAY参数表示在系统识别出用户操作之前，需要保持按压状态一段时间。

11.5 在Windows Embedded CE 6.0中实现鼠标点击绘制

前面介绍了Windows Embedded CE 6.0中的一些输入方式，如鼠标输入、键盘输入等。在这里将通过一个程序实例介绍如何在Windows Embedded CE中实现对鼠标点击事件的响应。

11.5.1 程序实例中涉及的主要函数

要在鼠标点击绘制的程序中实现如下的功能：

1）程序运行的时候，会首先在屏幕上绘制出4×4大小的格子。

2）当用鼠标单击每一个格子的时候，会在上面绘制出相应的图形。

3）单击鼠标的时候，首先绘制三角形，再次单击的时候绘制椭圆形；然后按照此顺序交替进行绘制。

4）在画面的中央会显示出下一个将要绘制图形的提示，如将要绘制的是三角形还是椭圆形。

那么在Windows Embedded CE 6.0系统中，是如何通过Visual C++ 2005来实现上述功能的

呢？下面就来看一下。

在开始介绍各个相关函数以前，先来看一下如何实现消息和函数的对应。从本章前面的内容介绍中会看到，当按下鼠标左侧按键的时候会激发WM_LBUTTONUP消息，而当屏幕绘制的时候则会激发WM_PAINT消息。在Embedded Visual C++中是通过下面方式实现消息和函数对应的：

```
const struct decodeUINT MainMessages[] = {
WM_PAINT, DoPaintMain,
WM_LBUTTONUP, DoLButtonUpMain,
WM_DESTROY, DoDestroyMain,};
```

也就是说，当激发WM_PAINT消息的时候，调用DoPaintMain函数；当激发WM_LBUTTONUP的时候，调用DoLButtonUpMain函数；当激发WM_DESTROY消息的时候，会调用DoDestroyMain函数。这样就实现了对不同消息的相应处理。例如，在程序中单击鼠标左键的时候，就会引发WM_LBUTTONUP消息，也就会调用DoLButtonUpMain函数。这也是本程序框架构成的核心所在。

在这里需要特别说明的是，当程序刚刚启动的时候，也是会激发WM_PAINT消息的，因此如果想要在程序初始化的时候就绘制出一些图形，就需要在DoPaintMain函数中进行添加。会在代码中看到在DoPaintMain函数中添加了DrawBoard函数，也就是实现对4×4格子的绘制。

为了实现对画面中间位置提示信息的交替显示，在这里引入了一个变量bTurn。当单击一次鼠标左键的时候，就会改变一次bTurn的值，这样bTurn的值就会在0和1之间来回地切换，在每次调用DoPaintMain函数的时候，就会根据bTurn值的不同，进行不同信息的显示。同样的道理，用一个全局类型的数组Temp来记录每一个格子中应该绘制图形的类型。正是通过这种方式，实现了文字提示信息和绘制图形类型的循环交替。

当单击鼠标左键的时候，除了会做出一系列设置以外，还应特别注意下面这句代码：

```
InvalidateRect (hWnd, NULL, FALSE);
```

这句代码可以重新激发一个WM_PAINT事件，同时背景区域又不会被重新绘制。感兴趣的读者可以把InvalidateRect中的最后一个参数修改成TRUE，运行一下程序看看会有什么现象发生。我们会看到，背景画面出现了闪烁现象，这是因为设置成TRUE以后，整个背景画面都进行了重新绘制，因此画面看上去会有闪烁感。如果去掉了InvalidateRect这一行代码，就会无法重新激发WM_PAINT事件，也就无法实现单击屏幕时绘制相应的图形了。如果有兴趣，可以尝试去掉这个InvalidateRect来验证一下。

11.5.2 程序运行效果

程序启动后的画面如图11-4所示。

图11-5展示了程序运行中的画面。

11.5.3 程序源代码——ClickDraw.h

```
#define dim(x) (sizeof(x) / sizeof(x[0]))

struct decodeUINT {                              // Structure associates
    UINT Code;                                   // messages
                                                 // with a function.
```

```
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};

//----------------------------------------------------------------------
// Function prototypes
//
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

// Window procedures
LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

// Message handlers
LRESULT DoSizeMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoPaintMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoLButtonDownMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoLButtonUpMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);

// Game function prototypes
void DrawPic (HDC hdc, HPEN hPen, RECT *prect, INT nCell, INT nType);
void DrawBoard (HDC hdc, RECT *prect);
```

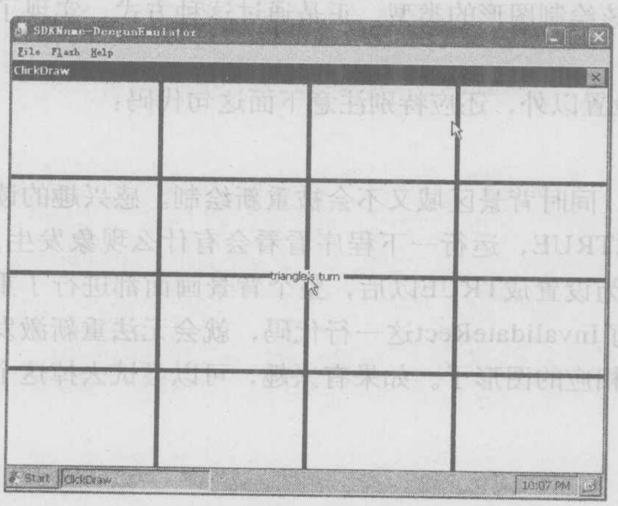


图11-4 程序启动后的画面

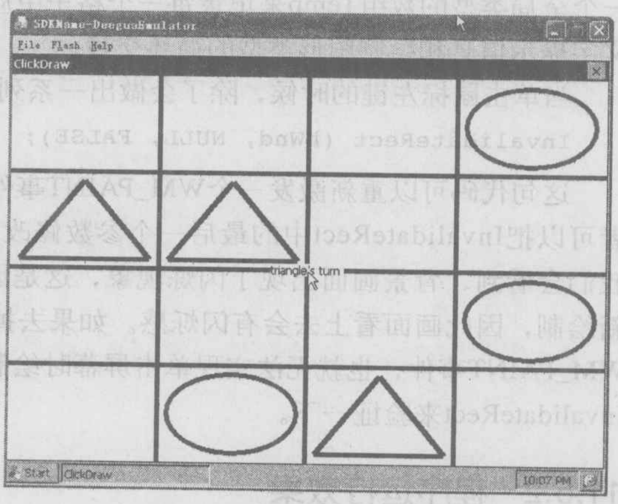


图11-5 程序运行中的画面

11.5.4 程序源代码——ClickDraw.cpp

```
#include <windows.h>            // For all that Windows stuff
#include <commctrl.h>           // Command bar includes
#include "ClickDraw.h"          // Program-specific stuff

//----------------------------------------------------------------------
// Global data
//
```

```
const TCHAR szAppName[] = TEXT ("ClickDraw");
HINSTANCE hInst;
RECT rectBoard ;
BYTE Temp[16];
BYTE bTurn = 0;

const struct decodeUINT MainMessages[] = {
    WM_PAINT, DoPaintMain,
    WM_LBUTTONUP, DoLButtonUpMain,
    WM_DESTROY, DoDestroyMain,
};

//======================================================================
//
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    HWND hwndMain;

    // Initialize this instance.
    hwndMain = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;
    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//----------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow) {
    WNDCLASS wc;
    HWND hWnd;

    // Save program instance handle in global variable.
    hInst = hInstance;

#if defined(WIN32_PLATFORM_PSPC) || defined(WIN32_PLATFORM_WFSP)
    // If Windows Mobile, allow only one instance of the application.
```

```
        hWnd = FindWindow (szAppName, NULL);
        if (hWnd) {
            SetForegroundWindow ((HWND)(((DWORD)hWnd) | 0x01));
            return 0;
        }
    #endif
        // Register application main window class.
        wc.style = 0;                                  // Window style
        wc.lpfnWndProc = MainWndProc;                  // Callback function
        wc.cbClsExtra = 0;                             // Extra class data
        wc.cbWndExtra = 0;                             // Extra window data
        wc.hInstance = hInstance;                      // Owner handle
        wc.hIcon = NULL,                               // Application icon
        wc.hCursor = LoadCursor (NULL, IDC_ARROW);// Default cursor
        wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
        wc.lpszMenuName =  NULL;                       // Menu name
        wc.lpszClassName = szAppName;                  // Window class name

        if (RegisterClass (&wc) == 0) return 0;

        // Create main window.
        hWnd = CreateWindowEx (WS_EX_NODRAG, szAppName, TEXT ("ClickDraw"),
                             WS_VISIBLE | WS_CAPTION | WS_SYSMENU,
                             CW_USEDEFAULT, CW_USEDEFAULT,
                             CW_USEDEFAULT, CW_USEDEFAULT,
                             NULL, NULL, hInstance, NULL);
        // Return fail code if window not created.
        if (!IsWindow (hWnd)) return 0;

        // Standard show and update calls
        ShowWindow (hWnd, nCmdShow);
        UpdateWindow (hWnd);
        return hWnd;
    }
    //----------------------------------------------------------------------
    // TermInstance - Program cleanup
    //
    int TermInstance (HINSTANCE hInstance, int nDefRC) {
        return nDefRC;
    }
    //======================================================================
    // Message handling procedures for MainWindow
    //
    //----------------------------------------------------------------------
    // MainWndProc - Callback function for application window
    //
```

```
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message. If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc(hWnd, wMsg, wParam, lParam);
}
//----------------------------------------------------------------------
// DoSizeMain - Process WM_SIZE message for window.
//


//----------------------------------------------------------------------
// DoPaintMain - Process WM_PAINT message for window.
//
LRESULT DoPaintMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                     LPARAM lParam) {
    PAINTSTRUCT ps;
    RECT rect;
    HFONT hFont, hOldFont;
    HDC hdc;

    GetClientRect (hWnd, &rect);
    rectBoard = rect;
    hdc = BeginPaint (hWnd, &ps);

    DrawBoard (hdc, &rect);

    hFont = (HFONT)GetStockObject (SYSTEM_FONT);
    hOldFont = (HFONT)SelectObject (hdc, hFont);

    if (bTurn == 0)
        DrawText (hdc, TEXT ("triangle's turn "), -1, &rect,
                  DT_CENTER | DT_VCENTER);
    else
        DrawText (hdc, TEXT (" Round's turn"), -1, &rect,
                  DT_CENTER | DT_VCENTER );

    SelectObject (hdc, hOldFont);
    EndPaint (hWnd, &ps);
    return 0;
```

```
}
//----------------------------------------------------------------------
// DoLButtonUpMain - Process WM_LBUTTONUP message for window.
//
LRESULT DoLButtonUpMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                          LPARAM lParam) {
    POINT pt;
    INT dx, dy, nCell = 0;

    pt.x = LOWORD (lParam);
    pt.y = HIWORD (lParam);

    if (PtInRect (&rectBoard, pt)){

        dx = (rectBoard.right - rectBoard.left)/4;
        dy = (rectBoard.bottom - rectBoard.top)/4;

        nCell = (pt.x / dx);

        nCell += (pt.y / dy) * 4;

        if (Temp[nCell] == 0) {
            if (bTurn) {
                Temp[nCell] = 2;
                bTurn = 0;
            } else {
                Temp[nCell] = 1;
                bTurn = 1;
            }
            InvalidateRect (hWnd, NULL, FALSE);
        } else {
            MessageBeep (0);
            return 0;
        }
    }
    return 0;

}
//----------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                        LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
```

```
void DrawPic (HDC hdc, HPEN hPen, RECT *prect, INT nCell, INT nType) {
    POINT pt[2];
    INT dx, dy;
    RECT rect;

    dx = (prect->right - prect->left)/4;
    dy = (prect->bottom - prect->top)/4;

    rect.left = (dx * (nCell % 4) + prect->left) + 10;
    rect.right = rect.right =  rect.left + dx - 20;
    rect.top = dy * (nCell / 4) + prect->top + 10;
    rect.bottom = rect.top + dy - 20;

    if (nType == 1) {
        pt[0].x = rect.left+(rect.right-rect.left)/2;
        pt[0].y = rect.top;
        pt[1].x = rect.left;
        pt[1].y = rect.bottom;
        Polyline (hdc, pt, 2);

        pt[1].x=rect.right;
        Polyline(hdc,pt,2);

        pt[0].x=rect.left;
        pt[0].y=rect.bottom;
        pt[1].x=rect.right;
        pt[1].y=rect.bottom;
        Polyline(hdc,pt,2);

    } else if (nType == 2) {
        Ellipse (hdc, rect.left, rect.top, rect.right, rect.bottom);
    }
    return;
}

//----------------------------------------------------------------------
// DrawBoard - Draw the tic-tac-toe board.
//  VK_MENU
void DrawBoard (HDC hdc, RECT *prect) {
    HPEN hPen, hOldPen;
    POINT pt[2];
    LOGPEN lp;
    INT i, dx, dy;

    lp.lopnStyle = PS_SOLID;
    lp.lopnWidth.x = 5;
    lp.lopnWidth.y = 5;
```

```
    lp.lopnColor = RGB (0, 0, 0);
    hPen = CreatePenIndirect (&lp);

    hOldPen = (HPEN)SelectObject (hdc, hPen);

    dx = (prect->right - prect->left)/4;
    dy = (prect->bottom - prect->top)/4;

    pt[0].x = dx + prect->left;
    pt[1].x = dx + prect->left;
    pt[0].y = prect->top;
    pt[1].y = prect->bottom;
    Polyline (hdc, pt, 2);

    for(i=0;i<2;i++)
    {
        pt[0].x += dx;
        pt[1].x += dx;
        Polyline (hdc, pt, 2);
    }

        pt[0].x = prect->left;
        pt[1].x = prect->right;
        pt[0].y = dy + prect->top;
        pt[1].y = dy + prect->top;
        Polyline (hdc, pt, 2);

        for(i=0;i<2;i++)
        {
        pt[0].y += dy;
        pt[1].y += dy;
        Polyline (hdc, pt, 2);
        }

        for (i = 0; i < dim (Temp); i++)
            DrawPic (hdc, hPen, &rectBoard, i, Temp[i]);

        SelectObject (hdc, hOldPen);
        DeleteObject (hPen);
        return;

}
```

11.6 在Windows Embedded CE 6.0中实现鼠标移动轨迹的绘制

在学习完前面的程序实例以后，再来看看如何在Windows Embedded CE 6.0设备中绘制鼠标的移动轨迹。

11.6.1 程序实例涉及的主要函数

在前面的程序实例中，需要把鼠标在移动过程中的点绘制出来，因此就需要在激发WM_MOUSEMOVE消息的时候进行相应的处理。和前面中采用的方法一样，在这里仍旧采用消息和函数对应的方式来进行消息处理。程序中的代码为：

```
const struct decodeUINT MainMessages[] = {
WM_MOUSEMOVE, MouseMain,
WM_DESTROY, DoDestroyMain,};
```

从上面的代码中可以清晰地看到，对鼠标移动时相关功能的实现都是在MouseMain函数中完成的。

在这个程序实例中，使用了两种方式进行鼠标轨迹的绘制。一种是直接把鼠标移动位置的坐标点描述出来，一种是使用GetMouseMovePoints函数，先存储鼠标移动过的位置的坐标，再把它们绘制出来。在这里之所以采用了以上两种方式，是因为直接绘制坐标点会存在一些问题。例如，当鼠标移动的速度很快时，Windows CE设备由于硬件性能上的限制会来不及发送所有的消息，在屏幕上看到的现象就是两个点之间有很大的空隙。为了解决这个问题，在Windows CE中系统又提供了另外一个函数GetMouseMovePoints来帮助程序开发人员捕获手写笔的轨迹（在这里用鼠标的移动来模拟手写笔）。

为了观察的更为清晰，采用直接绘制坐标方法绘制的点用红色来表示，当在移动鼠标的同时，若还按下Shift按键，就会在屏幕上绘制出黑色的点，该点为采用GetMouseMovePoints方法绘制的。

和GetMouseMovePoints方法相关的代码段为：

```
if (wMsg == WM_MOUSEMOVE) {
    if (wParam & MK_SHIFT)
        GetMouseMovePoints (pt, 100, &uPoints);

    for (i = 0; i < uPoints; i++) {
        pt[i].x /= 4;
        pt[i].y /= 4;

        MapWindowPoints (HWND_DESKTOP,hWnd ,&pt[i] ,1);
        SetPixel (hdc, pt[i].x,   pt[i].y, RGB (25, 25, 25));
        SetPixel (hdc, pt[i].x+1, pt[i].y,  RGB (25, 25, 25));
        SetPixel (hdc, pt[i].x,   pt[i].y+1,  RGB (25, 25, 25));
        SetPixel (hdc, pt[i].x+1, pt[i].y+1,  RGB (25, 25, 25));
    }
}
```

其含义为当鼠标（手写笔）移动的同时若按下Shift按键，首先使用GetMouseMovePoints函数记录下移动的点，然后把这些点进行坐标计算和转换，最后使用SetPixel函数把这些点绘制出来。

在这里有几点需要特别指出，程序中把各个点的坐标除以4是因为GetMouseMovePoints函数是按照触摸板的分辨率返回点的，而不是按照屏幕的分辨率。同时触摸板的分辨率通常设置为屏幕分辨率的4倍，因此需要把GetMouseMovePoints函数返回的坐标除以4来转换成屏幕的坐标。

在上面这段代码中还使用了MapWindowPoints函数，这个函数的函数原型为：

```
int MapWindowPoints (HWND hWndFrom,  HWND hWndTo,
                     LPPOINT lpPoints,   UINT cPoints );
```

这个函数的功能是将一个窗口客户区坐标的点转换到另一窗口的客户区坐标系统中。MapWindowPoints函数各个参数的含义为：

- hWndFrom：参数hWndFrom为原坐标的窗口。
- hWndTo：参数hWndTo为目的坐标的窗口。
- lpPoints：参数lpPoints为指向POINT结构数组的指针。
- cPoints：参数cPoints表示参数lpPoints指向的POINT结构的数量。

感兴趣的读者可以尝试去掉MapWindowPoints这一行代码，执行程序以后看看会有什么现象发生。这个现象将进一步加深您对MapWindowPoints函数功能的理解。

细心的读者一定还会从上面的代码中看到在绘制点的时候使用了下面的代码：

```
SetPixel (hdc, pt[i].x,   pt[i].y, RGB (25,  25,  25));
            SetPixel (hdc,   pt[i].x+1, pt[i].y,   RGB (25, 25, 25));
```

其中SetPixel的作用是在指定位置用指定颜色绘指出像素点。其函数原型为：

```
COLORREF SetPixel (HDC hdc,   int X,   int Y, COLORREF crColor);
```

各个参数的含义为：

- hdc：参数hdc表示设备句柄。
- X：参数X表示绘制位置的X坐标。
- Y：参数Y表示绘制位置的Y坐标。
- crColor：参数crColor表示采用的颜色。

在这里之所以采用了pt[i].x+1的方式，是为了在坐标点的周围再绘制几个点，这样使得绘制的点看上去“大”一点，方便读者看清楚显示效果。在程序后面的直接绘制像素的代码中，也采用了这一绘制原则。其代码为：

```
SetPixel (hdc, PT.x, PT.y, RGB (255, 0, 0));
SetPixel (hdc, PT.x+1, PT.y, RGB (255, 0, 0));
SetPixel (hdc, PT.x, PT.y+1, RGB (255, 0, 0));
SetPixel (hdc, PT.x+1, PT.y+1, RGB (255, 0, 0));
```

11.6.2 程序运行效果

图11-6展示了程序运行时的效果。

图11-7展示了程序运行中的效果。

11.6.3 程序源代码——PenTest.h

```
#define dim(x) (sizeof(x) / sizeof(x[0]))

struct decodeUINT {
    UINT Code;

    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
```

```
};
struct decodeCMD {
    UINT Code;
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);
};


HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

LRESULT DoPaintMain (HWND, UINT, WPARAM, LPARAM);
LRESULT MouseMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);
```

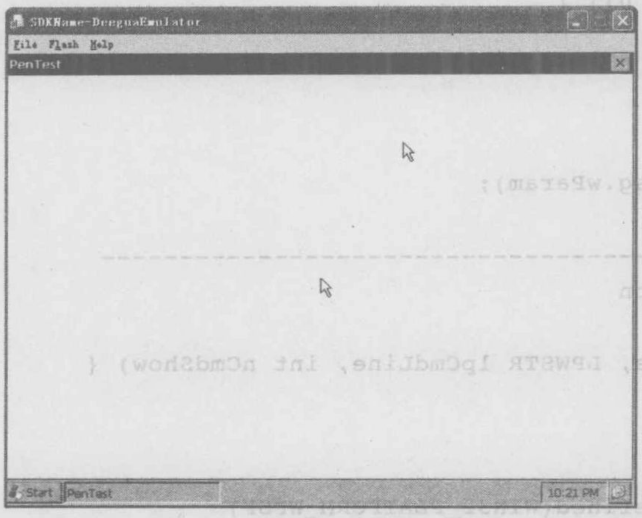


图11-6 程序启动时的效果

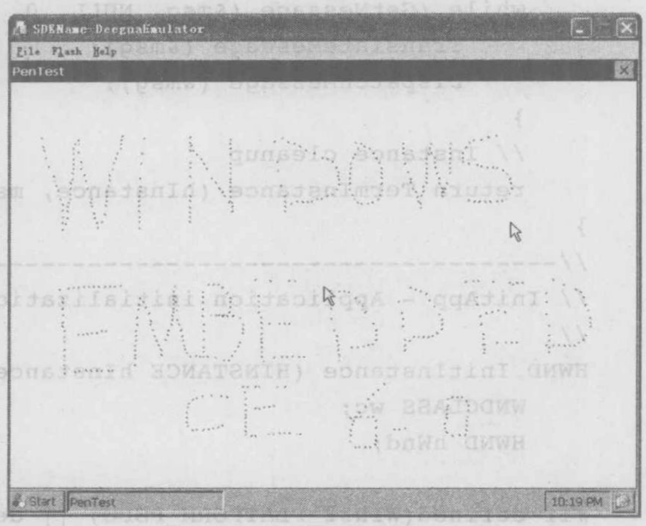


图11-7 程序运行中的效果

11.6.4 程序源代码——PenTset.cpp

```
#include <windows.h>                    // For all that Windows stuff
#include "PenTest.h"                    // Program-specific stuff

//----------------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT ("PenTest");
HINSTANCE hInst;

const struct decodeUINT MainMessages[] = {
    WM_MOUSEMOVE, MouseMain,
    WM_DESTROY, DoDestroyMain,
};
```

```
//======================================================================
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    int rc = 0;
    HWND hwndMain;

    // Initialize this instance.
    hwndMain = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;

    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//----------------------------------------------------------------------
// InitApp - Application initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow) {
    WNDCLASS wc;
    HWND hWnd;

#if defined(WIN32_PLATFORM_PSPC) || defined(WIN32_PLATFORM_WFSP)
// If Windows Mobile, allow only one instance of the application.
    hWnd = FindWindow (szAppName, NULL);
    if (hWnd) {
        SetForegroundWindow ((HWND)(((DWORD)hWnd) | 0x01));
        return 0;
    }
#endif
    // Save program instance handle in global variable.
    hInst = hInstance;

    // Register application main window class.
    wc.style = 0;                             // Window style
    wc.lpfnWndProc = MainWndProc;             // Callback function
    wc.cbClsExtra = 0;                        // Extra class data
    wc.cbWndExtra = 0;                        // Extra window data
    wc.hInstance = hInstance;                 // Owner handle
    wc.hIcon = NULL,                          // Application icon
```

```
    wc.hCursor = LoadCursor (NULL, IDC_ARROW);// Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                   // Menu name
    wc.lpszClassName = szAppName;              // Window class name

    if (RegisterClass (&wc) == 0) return 0;

    // Create main window.
    hWnd = CreateWindowEx (WS_EX_NODRAG, szAppName, TEXT ("PenTest"),
                          WS_VISIBLE | WS_CAPTION | WS_SYSMENU,
                          CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, NULL, NULL, hInstance, NULL);
    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
//----------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {
    return nDefRC;
}
//======================================================================
// Message handling procedures for MainWindow
//

//----------------------------------------------------------------------
// MainWndProc - Callback function for application window
//
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//----------------------------------------------------------------------
```

```
// DoMouseMain - Process WM_LBUTTONDOWN and WM_MOUSEMOVE messages
// for window.
//
LRESULT MouseMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                    LPARAM lParam) {
    POINT pt[80];
    POINT PT;
    UINT i, uPoints = 0;
    HDC hdc;

    PT.x = LOWORD (lParam);
    PT.y = HIWORD (lParam);
    hdc = GetDC (hWnd);

    if (wMsg == WM_MOUSEMOVE) {
        if (wParam & MK_SHIFT)
            GetMouseMovePoints (pt, 100, &uPoints);

        for (i = 0; i < uPoints; i++) {
            pt[i].x /= 4;
            pt[i].y /= 4;

            MapWindowPoints (HWND_DESKTOP,hWnd ,&pt[i] ,1);
            SetPixel (hdc, pt[i].x,   pt[i].y, RGB (25, 25, 25));
            SetPixel (hdc, pt[i].x+1, pt[i].y,  RGB (25, 25, 25));
            SetPixel (hdc, pt[i].x,   pt[i].y+1,  RGB (25, 25, 25));
            SetPixel (hdc, pt[i].x+1, pt[i].y+1,  RGB (25, 25, 25));
        }
    }

    SetPixel (hdc, PT.x, PT.y, RGB (255, 0, 0));
    SetPixel (hdc, PT.x+1, PT.y, RGB (255, 0, 0));
    SetPixel (hdc, PT.x, PT.y+1, RGB (255, 0, 0));
    SetPixel (hdc, PT.x+1, PT.y+1, RGB (255, 0, 0));
    ReleaseDC (hWnd, hdc);

    Sleep(10);
    return 0;
}

LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
```

第12章 Windows Embedded CE 6.0版“连连看”游戏程序的设计

前面已经介绍了使用Visual C++在Windows Embedded CE 6.0中实现屏幕点击绘制等程序的基本步骤，在这里将要介绍一下在Visual Studio 2005中如何使用Visual C#设计Windows Embedded CE 6.0版“连连看”程序的步骤。通过本章不但会学习到整个程序设计的过程，还会了解到Windows Embedded CE 6.0程序设计与一般PC程序开发的不同，使读者通过一步一步的具体操作，亲身体会一个“小型”程序的开发流程。

因为本程序采用了面向对象的程序设计思想，因此在这里首先简要介绍一下什么是面向对象，该种程序设计方式具有哪些特点。在这里需要特别说明的是，由于本书的重点是介绍如何开发Windows Embedded CE 6.0的相关程序，因此有关C++、C#等基本语法知识不做重点讲解，相关内容可参考其他书籍。

12.1 面向对象程序设计简介

面向对象方法（Object-Oriented Method）是一种把面向对象的思想应用于软件开发过程中，指导开发活动的系统方法，简称OO（Object-Oriented）方法，是建立在“对象”概念基础上的方法学。对象是由数据和容许的操作组成的封装体，与客观实体有直接对应关系，一个对象类定义了具有相似性质的一组对象。而继承性是对具有层次关系的类的属性和操作进行共享的一种方式。所谓面向对象就是基于对象概念，以对象为中心，以类和继承为构造机制，来认识、理解、刻画客观世界和设计、构建相应的软件系统。

面向对象是当前计算机界关心的重点，它是20世纪90年代软件开发方法的主流。面向对象的概念和应用已超越了程序设计和软件开发，扩展到很宽的范围，如数据库系统、交互式界面、应用结构、应用平台、分布式系统、网络管理结构、CAD技术、人工智能等领域。

首先来看一下传统软件设计方式具有哪些不足。

12.1.1 传统软件设计方法存在的问题

传统的软件设计方式主要存在以下几个问题。

1. 软件的重用性差

在这里提到的重用性是指同一事物不经修改或稍加修改就可多次重复使用的性质。软件重用性是软件工程追求的目标之一。如当推出的软件需要按照用户的新需求进行更改的时候，如果采用了面向对象程序设计方法，就可以只修改其中的一部分代码，而与之关联的其余代码模块就不用进行修改。这一点，可以在本书后面的程序实例讲解中具体地看到。

2. 软件可维护性差

软件工程强调软件的可维护性，强调文档资料的重要性，规定最终的软件产品应该由完整、一致的配置成分组成。在软件开发过程中，始终强调软件的可读性、可修改性和可测试性是软件重要的质量指标。实践证明，用传统方法开发出来的软件，维护时其费用和成本仍然很高，其原因是可修改性差，维护困难，导致可维护性差。

3. 开发出的软件不能满足用户需要

用传统的结构化方法开发大型软件系统涉及各种不同领域的知识，在开发需求模糊或需求动态变化的系统时，所开发出的软件系统往往不能真正满足用户的需要。

用结构化方法开发的软件，其稳定性、可修改性和可重用性都比较差，这是因为结构化方法的本质是功能分解，从代表目标系统整体功能的单个处理着手，自顶向下不断把复杂的处理分解为子处理，这样一层一层地分解下去，直到仅剩下若干个容易实现的子处理功能为止，然后再用相应的工具来描述各个最低层的处理。因此，结构化方法是围绕实现处理功能的“过程”来构造系统的。然而，用户需求的变化大部分是针对功能的，因此，这种变化对于基于过程的设计来说是灾难性的。用这种方法设计出来的系统结构常常是不稳定的，用户需求的变化往往造成系统结构的较大变化，从而需要花费很大代价才能实现这种变化。

12.1.2 面向对象程序方法的特点

与前面提到的结构化的程序设计方法相比，面向对象程序设计方式具有以下几个特点：

1. 对象唯一性

在面向对象程序设计方法中，每个对象都有自身唯一的标识，通过这种标识，可找到相应的对象。在对象的整个生命期中，它的标识都不改变，不同的对象不能有相同的标识。

2. 分类性

这里提到的分类性是指将具有一致的数据结构（属性）和行为（操作）的对象抽象成类。一个类就是这样一种抽象，它反映了与应用有关的重要性质，而忽略其他一些无关内容。任何类的划分都是主观的，但必须与具体的应用有关。

3. 继承性

继承性是子类自动共享父类数据结构和方法的机制，这是类之间的一种关系。在定义和实现一个类的时候，可以在一个已经存在的类的基础之上来进行，把这个已经存在的类所定义的内容作为自己的内容，并加入若干新的内容。我们会在本书后面的程序实例中看到很多类之间互相继承的例子。同时，继承性也是面向对象程序设计语言不同于其他语言的最重要的特点，是其他语言所没有的。

4. 多态性

多态性是指相同的操作或函数、过程可作用于多种类型的对象上并获得不同的结果。不同的对象，收到同一消息可以产生不同的结果，这种现象称为多态性。多态性允许每个对象以适合自身的方式去响应共同的消息，同时多态性还增强了软件的灵活性和重用性。

12.1.3 面向对象程序设计方法中涉及的基本概念

面向对象技术在使用过程中有许多概念，它在吸收以往计算机程序语言的同时，又给这些术

语添加了一些新的内容。本节主要简单介绍一些涉及的常用术语，以便使读者有一个直观印象。

1. 对象

对象是人们要进行研究的任何事物，从最简单的整数到复杂的飞机等均可看作对象，它不仅能表示具体的事物，还能表示抽象的规则、计划或事件。

2. 对象的状态和行为

对象具有状态，一个对象用数据值来描述它的状态；对象还有操作，用于改变对象的状态，对象及其操作就是对象的行为，对象实现了数据和操作的结合，使数据和操作封装于对象的统一体中。

3. 类

具有相同或相似性质的对象的抽象就是类。因此，对象的抽象是类，类的具体化就是对象，也可以说类的实例是对象。类具有属性，它是对象的状态的抽象，用数据结构来描述类的属性；类也具有操作，它是对象的行为的抽象，用操作名和实现该操作的方法来描述。

4. 封装

封装（encapsulation）是划分一个抽象的各个元素（它们组成这个抽象的结构和行为）的过程，它用来分割一个抽象和它实现的契约接口。封装的特点就是具有明显的边界和一个接口，所有对象的分层都在这个边界内，外部通过这个接口访问对象的内部成员。

5. 继承

继承（inheritance）是类之间的关系之一，其中一个类共享另一个或多个类中定义的结构或行为，其中一个子类从一个或多个更普遍的超类继承；子类通常通过增加或者重新定义已存在的结构和行为类特别指明它的超类。

继承有单继承和多继承之分，单继承是指子类仅从一个父类继承属性和方法，这个子类是可以覆盖父类的部分属性和方法的；多继承是指子类可以从多个父类继承属性和方法。继承的使用可以简化软件开发的复杂过程，简化类和对象之间的接口，实现软件的重复使用。

6. 多态性

所谓的多态性就是指一个名字（如变量声明）可以表示许多不同类（而它们有某个共同的超类）的对象。因此，任何对象都可以由它的名字来表示，并能以不同的方式响应某个共同的操作集。多态性在结构方面提供了灵活性，它使不同的对象在收到同一消息后，产生完全不同的结果。多态性增强了软件的灵活性和重用性，允许用更为明确、易懂的方式去建立通用软件，多态性和继承性相结合使软件具有更广泛的重用性和可扩充性。

7. 消息

消息（message）是指一个对象在另一个对象上执行的操作，message、method和operation三个术语可以互换。具体来说，消息是指对象之间在交互中所传送的通信信息。一个消息应该包含以下信息：消息名、接收消息对象的标识、服务标识、消息和方法、输入信息、回答信息。使对象之间互相联系、协同工作，实现系统的各种服务。

通常一个对象向另一个对象发送信息请求某项服务，接收对象响应该消息，激发所要求的服务操作，并将操作结果返回给请求服务的对象。消息传递是对象与外部世界关联的唯一途径，通过对象接收消息后执行某些方法，可以获取对象状态，进而改变对象状态，完成某项计算或控制

功能。可以通过方法调用或网络传输协议来实现消息的传递。但注意，在并发系统中，多个控制线程（Thread of Control）并发执行，情况就复杂得多，消息可以是发出服务请求、提交数据、发布事件信息或传递同步控制信息。

12.1.4 面向对象程序设计方法中对象的概念

日常生活中，我们会接触各式各样的事物，每一种事物，我们都可以把它看作一个对象，如一台电脑、一本书、一座建筑物等，还有一些抽象的事物也可以把它视为一个对象，如心理学的研究对象就是人的心理。这些对象都有一个明确的边界限定。此外，也有一些边界并不明确的对象，如河流、人群等，但是它们所具有的属性，也可以把它们和其他东西区别开来，所以也可以视为对象。

通常情况下，对象包含状态、行为和标识三个特征。其中，对象的状态主要包含对象的（通常是静态的）所有属性和这些属性的当前（通常是动态的）值，对象的行为是一个对象根据它的状态改变和消息传送所采取的行动和作出的反应，对象的状态代表它的行为的积累结果；对象的标识则是一个对象用来和所有其他对象相区分的特性。

12.1.5 面向对象程序设计方法中类的概念

类的概念和对象的概念是紧密相连的。面向对象技术中的类（class）又可以称为对象类(object class)，它指一组具有相同属性和运算的对象的抽象，一组具有相同数据结构和相同操作的对象的集合。因此，可以说，对象是一个存在于时间和空间中的具体实体，而类仅代表了一个抽象，一个对象的“本质”。换句话说，对象就是类的实例。

例如，我们说“学生”，它代表了整个学生共有的特征，如果要说“大学生”或者“小学生”，就是学生这个类中的对象。

类的实现主要包括在类的接口中定义的所有操作的实现，它的接口可以细分为三个部分：一是共有的（public），即所有客户都可以访问的声明；二是保护的（protected），即只有类自己、该类的子类和它的友元类才可以访问的声明；三是私有的（private），即只有类自己和它的友元类才可以访问的声明。不同的编程语言提供了这三种不同的混合方式，程序员可以根据实际需要来选择最适合自己的访问权限。

12.1.6 面向对象中的继承

前面已经介绍过了继承的含义和特征，在实际操作中，继承能自动地为一个类提供来自另一个类的操作和数据结构。在这里，补充一点关于类的分类的知识。

类可以分为基类（或称父类）、派生类（或称子类）。在理论分析中，我们运用抽象原则，将对象的特殊性舍弃，提炼出对象的共同性，得到一个对象集的类，这个类就是这个对象集的基类，如本书后面程序实例中的Object类就是基类。

在此基础上，如果再考虑被舍弃的对象的特殊性，又会产生新类，我们将其称为派生类。如类“学生”，可以将类“大学生”和“中学生”看作是它的派生类，而类“学生”是它们的基类。派生类具有基类的全部特征，同时还具有自己的特征，它和基类有继承关系，可以将派生类看作基类的子集，如上面说的基类“学生”和派生类“大学生”的关系。

在上面的例子中，看到了继承的作用，它使我们可以用一种简单的方式来描述事物。图12-1所示的关于车辆类层次的说明，展示了继承的简单原理。

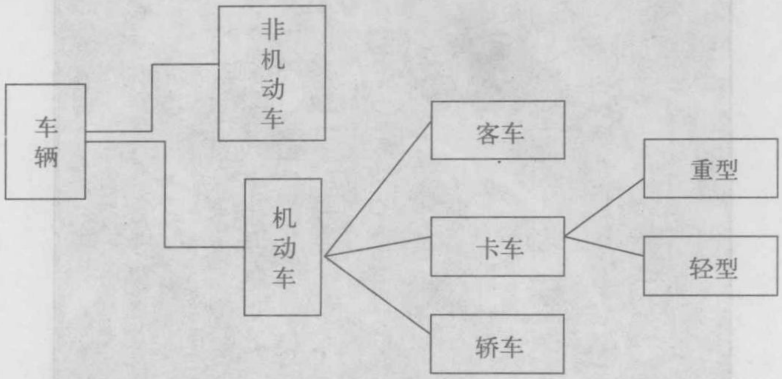


图12-1　车辆继承关系图

从图12-1中可以看到，车辆包含了机动车和非机动车，而机动车又包含轿车和卡车等。如果将“机动车”作为基类，则“轿车”和“卡车”就是“机动车”的派生类，而“车辆”则可以成为基类“机动车”的基类。所以，基类可以有基类，派生类也可以有派生类。

如果对图12-1比较感兴趣，可以在学习完本书后面的程序实例以后，自己尝试绘制一下这个继承关系图，看看在“连连看”游戏程序中各个类是怎样的一种继承关系。

12.2　“连连看”游戏程序介绍

“连连看”是目前非常流行的一款休闲类游戏，由于它游戏规则简单，画面精美，操作简便，而俘获了很多年轻人的心。甚至在上班族的工作时间里，很多职员也都忙里偷闲，完全沉浸在这款游戏的魅力中。

在游戏开始运行以后，我们只需要在多种图形中，按照游戏的规则（两个图形的连线中不能有别的图形遮挡，并且两个图形之间最多只允许有两个拐点）找出两个图案相同的图形，然后用鼠标把两者连接在一起，就可以消去彼此相连的两个图案。在规定的时间内，玩家消掉的图案越多，其得分也就越高。

图12-2展示了“连连看”游戏程序在自定义的Windows Embedded CE 6.0模拟器中启动时的画面，图12-3展示了该程序运行中的画面。

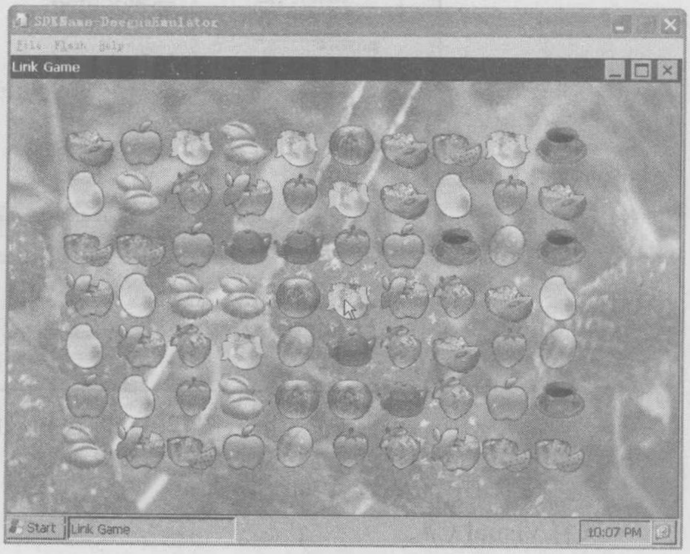


图12-2　程序启动时的画面

在看完了上面的两幅游戏程序截图以后，是不是觉得画面很吸引人呢？下面就来一起学习一下这款游戏的制作过程。

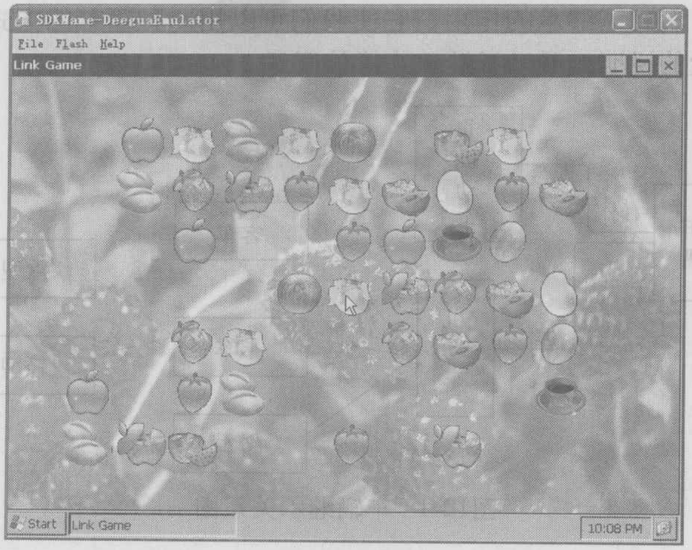


图12-3 程序运行中的画面

12.3 “连连看”游戏程序的设计步骤

12.3.1 程序工程的构建

首先启动Visual Studio 2005，选择File→New→Project。此时会出现图12-4所示的画面。

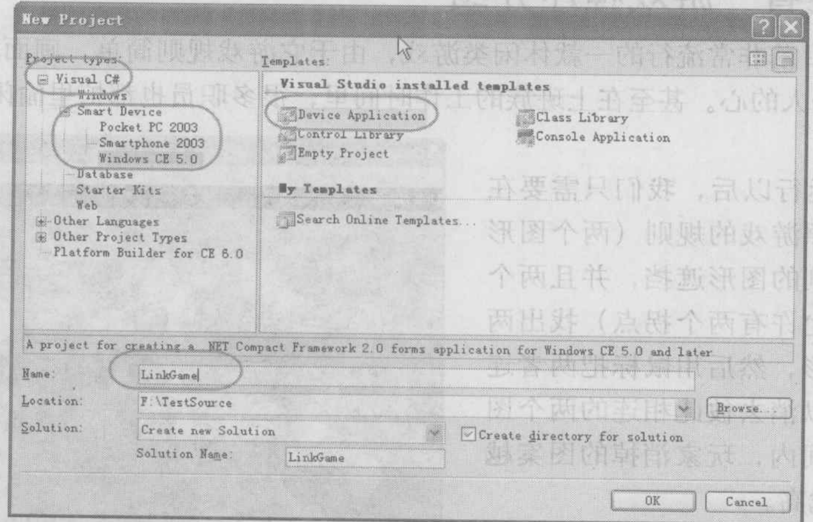


图12-4 构建相应的工程文件

在这个画面中需要进行如下设置：

- 选择Visual C#。
- 选择Smart Device中的Windows CE 5.0。
- 选择Device Application。
- 在Name文本框中输入工程文件的名称LinkGame。

单击OK按钮，这样就完成了相应的设置，此过程参见图12-4所示的画面。设置好以后会出现

图12-5所示的画面。

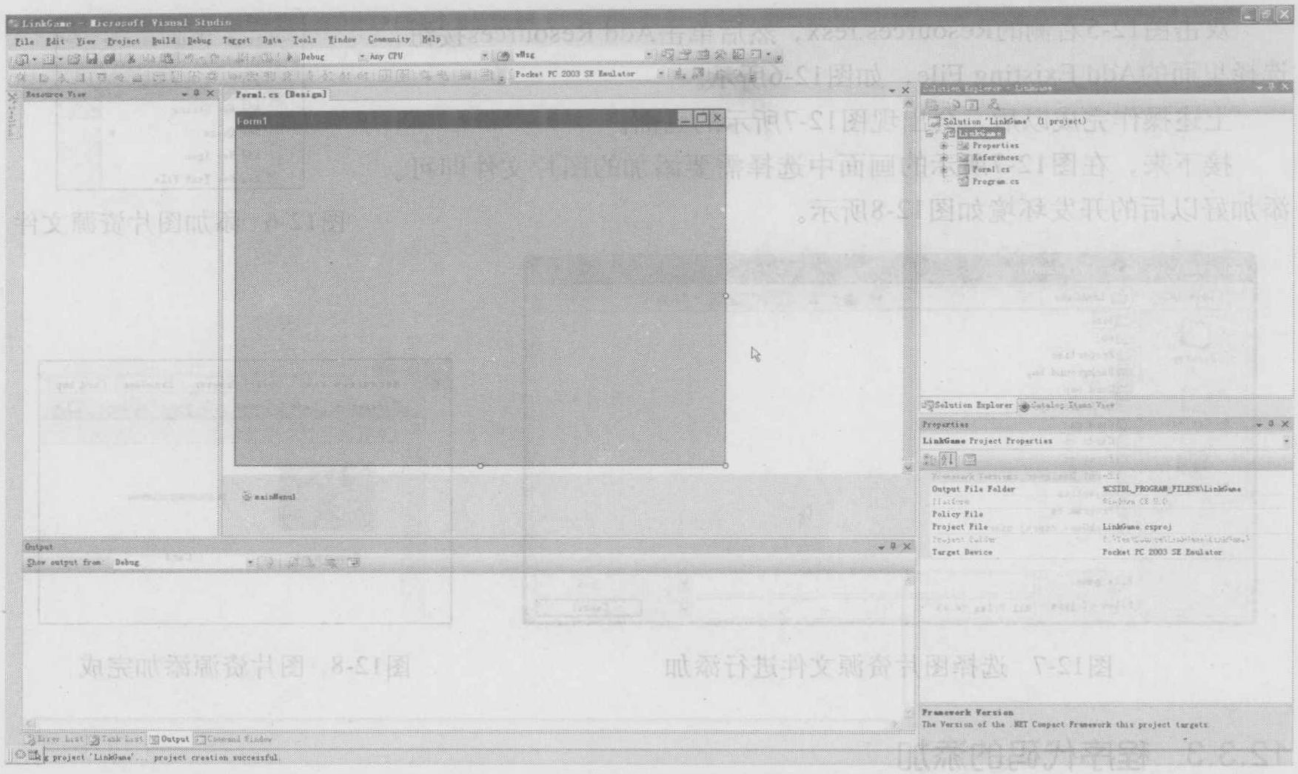


图12-5 工程文件构建完成

在图12-5所示的画面中，在Solution Explorer中双击Program.cs文件，会在代码显示界面看到下面一段代码：

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
namespace LinkGame
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [MTAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
    }
}
```

从代码注释中会看到Main函数是整个函数的入口函数，即程序是从这个Main函数开始执行的。

12.3.2 程序画面的添加

双击图12-5右侧的Resources.resx，然后单击Add Resources按钮，选择里面的Add Existing File，如图12-6所示。

上述操作完成以后，会出现图12-7所示的画面。

接下来，在图12-7所示的画面中选择需要添加的图片文件即可。添加好以后的开发环境如图12-8所示。

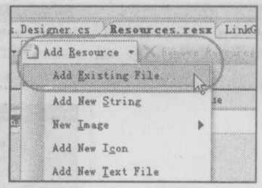


图12-6 添加图片资源文件

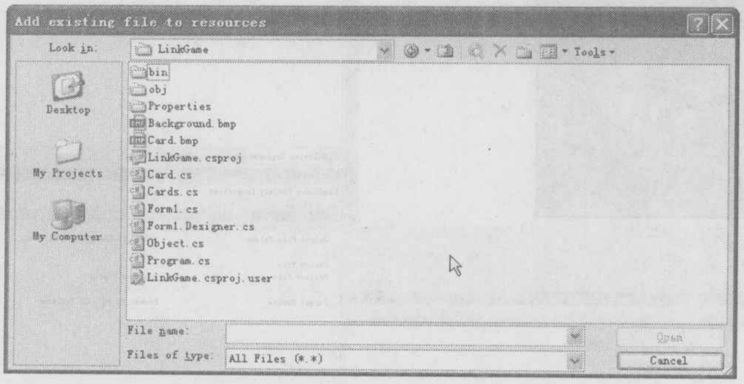


图12-7 选择图片资源文件进行添加

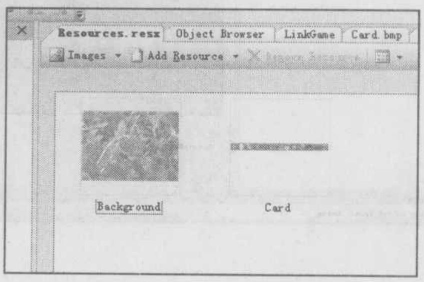


图12-8 图片资源添加完成

12.3.3 程序代码的添加

在这个游戏程序中，我们设置了所有对象的基类Object。在该程序中，与图形绘制相关的位置坐标、选择图形区域等操作都放在了这个类中。之所以这样做，是为了使整个程序看上去结构非常的清晰，同时也很好地体现了面向对象的程序设计理念，即把共性的东西抽象出来，整合在一起，以方便其余部分对其的调用实现。

在这个游戏程序中，我们还设置了卡片类Card。这个类继承自基类Object，因为卡片类中包含了卡片绘制等一些基本的功能，因此从这个意义上讲，继承自Object是最为合适的。在这里也可以看出，以后只需要修改基类Object的相关代码，就可以实现用不同图形的卡片绘制（如选择不同的卡片作为背景进行绘制等）。

因为游戏中会有很多卡片同时出现在画面上，因此我们还设置了卡片集合类Cards。既然被称为卡片集合类，顾名思义，该类主要负责和卡片集合相关的事项，如绘制整个卡片数组、实现背景卡片的随机排序、处理鼠标在卡片集合上的点击等。可以说，游戏中关键的功能，如处理鼠标点击、判断两张卡片之间是否可以连通等功能都是在这里实现的。

在了解完基类、卡片类、卡片集合类以后，接下来就来看一下该游戏程序的主类Program.cs。Program.cs中主要包含了游戏的主循环、应用程序的初始化、对象的处理、对象的绘制等游戏程序的主要功能。概括地讲，该游戏程序都是依靠这个类“串”起来的。因此建议读者要首先了解该类中如何把游戏程序的各个功能组合在一起，在此基础上再来仔细研究其余类中代码的具体实现。

关于代码的具体内容，请参考本书的附带光盘。

12.4 基类Object源代码详解

在程序代码中提到的Object类是所有对象的基类。在这个游戏中，图形的绘制、选中标示的绘制等都需要计算目的位置的坐标、源图像的坐标等。因此根据面向对象的思想，把所有这些经常用到的属性都抽象成了一个基类。在这个基类中包含了一些基本的操作，这样当再派生出具体类（如背景卡片）的时候，只需要继承自这个基类就可以了。这就是Object类的作用。

接下来，看看Object类中开始两行代码的含义。

```
using System.Drawing;
using System.Drawing.Imaging;
```

using在这里的作用是导入在其他命名空间中定义的类型。其中的System.Drawing命名空间提供了对 GDI+ 基本图形功能的访问；System.Drawing.Imaging 命名空间则提供高级 GDI+ 图像处理功能。所谓的GDI+就是GDI（Windows 早期版本中使用的图形设备接口）的后续版本，是Windows XP操作系统中的图形显示技术。它已经被集成到了.Net开发环境中，所以不管使用的操作系统是什么版本的，只要安装了.NET框架，就有了GDI+。

细心的读者一定会发现，在Object类中，我们把所有的成员变量都设置成了保护类型（protected）。在了解这样做的原因以前，先来了解一下修饰符public、protected和private的作用。

- public：指可以在任何地方被访问，包括类的外部。
- protected：指可以在它所属的类的内部被访问，或在派生于该类的类型中被访问。
- private：指只能在它所属的类的内部被访问。

从上面列出的修饰符介绍中可以看到，当把成员变量设置成保护类型以后，继承自Object类的子类不但可以使用这里面的成员变量，同时又实现了对外部的封装，有效地避免了私有（private）类型子类无法访问成员函数，公共（public）类型无法实现封装的弊病。

12.4.1 成员变量的含义

下面就来看一下函数中成员变量的含义。尽管在本章前面的代码展示中已经标注了变量的含义，但是在这里将更为详细地对其进行介绍。

表12-1列出了这些成员变量的含义。

表12-1 成员变量说明

名 称	含 义
game	对象所属的游戏。在程序代码中，通过这个成员函数，可以很容易地访问Game类中的Random函数等，而不用在每一个类中都把Random重写一遍。提高了程序的可读性和代码编写的效率
left	每一个对象左上角的X坐标，也就是对象左上角距离屏幕最左边的距离。这里的对象可以是卡片、背景图像等
top	每一个对象左上角的Y坐标，也就是对象左上角距离屏幕最顶端的距离。这里的对象可以是卡片、背景图像等
width	对象的宽度。通过这个变量，就可以计算出对象右下角的X坐标
height	对象的高度。通过这个变量，就可以计算出对象右下角的Y坐标
visible	对象是否可见。如当卡片被消掉以后，就会从屏幕上面消失。此时visible=false

（续）

名　称	含　义
image	绘制图像所使用的源图像。如绘制卡片的时候，是从图片资源Card.bmp里面选取图片的。这里的Card.bmp就是源图像
imageAttr	指定源图像的透明色信息，使其在程序中不被显示。在本程序中，粉色是不被显示的
srcX	源图像左上角的X坐标
srcY	源图像左上角的Y坐标

12.4.2 关于基类的构造函数

在一个类中，首先需要实现的就是该类的构造函数。下面就来看看下面几个构造函数：

```
public Object(Game game)
{
    this.game = game;
}

public Object(Game game, Image image, ImageAttributes imageAttr)
    : this(game)
{
    this.image = image;
    this.imageAttr = imageAttr;
}
```

细心的读者一定已经发现，上面这两个构造函数的函数名称是完全一样的，都是Object，只有里面的参数各不相同。我们把这种在同一个类中函数名称相同，而参数类型不同的现象叫做函数的重载。使用重载函数可以使程序代码在编译的时候，根据函数参数的不同自动地选择相应的函数进行编译。这样就可以大大降低程序的复杂程度。

在this.game=game、this.image=image和this.imageAttr=imageAttr中，this后面紧跟的game、image和imageAttr为Object类中的变量，在它们的前面加上this就表示是Object类中的变量；等号后面的那些值则为参数列表中传递进来的参数。

在第2个构造函数的参数列表后面“：this（game）”的作用是调用Object（Game game）。可以把this看作是Object，这样this（game）自然就可以被看成是Object（game），也就是调用第1个构造函数了。这样做的好处在于，当对Object（Game game）进行修改以后，第2个构造函数是不需要修改的。当然也可以在第2个构造函数中添加上this.game=game语句，虽然这样也可以实现原来函数的功能，但是如果第1个构造函数发生了变化，第2个构造函数也需要进行相应的修改。

12.4.3 基类中其余函数代码的解释

```
public bool Visible
{
    get
    {
        return this.visible;        //返回对象是否可见
    }
}
```

当卡片被消去以后，就处于不可见的状态，反之则处于可见状态。这也就是Visible函数的作用。

```
public virtual void Process()      //对对象进行相应的处理和操作。默认值为空。需要的时
                                   //候可以在子类重写该函数
        {
        }
```

而在上面这段代码中的Process函数，实现的是对对象进行相应的处理和操作。默认值为空。尽管在这里该函数没有实现任何功能，但是当需要进行相应功能扩展的时候，就可在子类中重写该函数。通过这种方式，就大大地提高了该游戏程序的可扩展性。这也是程序设计中一个非常重要的思想。

在这里还有一点需要注意的是，上面代码中的Process()被声明成了虚函数，如果想了解更多的关于虚函数的具体介绍，可参考本书前面的语法介绍部分。

```
public virtual void Draw(Graphics gx)  // 绘制对象
{
```

上面Draw函数中的参数Graphics是C#中的一个类，这个类在命名空间System.Drawing中被定义。Graphics主要提供了将对象绘制到屏幕的一些方法。

再来看看下面这段代码：

```
if (this.visible)                  //当对象是可见的时候
{
    gx.DrawImage(                  //调用DrawImage进行图像绘制
        this.image,
        new Rectangle(this.left, this.top, this.width, this.height),
        this.srcX,
        this.srcY,
        this.width,
        this.height,
        GraphicsUnit.Pixel,
        this.imageAttr);
    }
}
```

在上面的代码中，用到了DrawImage这个API函数，下面就来看一下DrawImage在C#中的用法。DrawImage的函数原型为：

```
public void DrawImage (Image image,
                       Rectangle destRect,
                       int srcX,
                       int srcY,
                       int srcWidth,
                       int srcHeight,
                       GraphicsUnit srcUnit,
                       ImageAttributes imageAttrs,
                       );
```

各个参数的含义为：

- image：所绘图像的源图像，在本程序中为图片资源里面的Card.bmp文件。

- destRect：它指定了所绘制图像的位置和大小。它可以将图像进行缩放以适合该矩形。
- srcX：源图像左上角的X坐标。
- srcY：源图像左上角的Y坐标。
- srcWidth：要采用的源图像的宽度。
- srcHeight：要采用的源图像的高度。
- srcUnit：用来指定源矩形度量单位，在本程序中为Card.bmp文件的度量单位。
- imageAttrs：指定对象的颜色属性，在本程序中指定粉色为透明色。

根据上面的解释就可以知道，在本程序中，DrawImage（……）的含义为：把Card.bmp文件中，以（srcX,srcY）为起始坐标，以像素点为单位，以宽为width，高为height的图片绘制到指定的矩形中。这个指定矩形的起始坐标为（left，top），它的宽为width，长为height。同时还指定了里面的颜色属性。

而下面这段代码：

```
public virtual void Reset()
{
}
```

它实现的功能是重置对象，其默认值为空。同在前面刚刚提到的Process（）函数一样，根据游戏程序功能的实际需要，可以在子类中重写该函数，以此实现对游戏程序的良好扩展。

```
public void Move(int left, int top)     // 移动对象
{
    this.left = left;                   //让对象左上角的X坐标为传递进来的参数的值
    this.top = top;                     //让对象左上角的Y坐标为传递进来的参数的值
}

public void Resize(int width, int height)  // 改变对象大小
{
    this.width = width;                 //让对象的宽为传递进来的参数的值
    this.height = height;               //让对象的高为传递进来的参数的值
}
```

当选择不同的卡片的时候，会用到Move函数，当改变时间条长短的时候，会用到Resize函数（这两点在后面的版本中会涉及）。在这里有一点需要特别指出的是，即使后面的扩充版本没有用到Move和Resize函数，也可以把这它们写在这里。其作用为基类的一个接口，方便以后对程序进行扩充，这也是面向对象思想的一个重要体现。

再来看看下面这段负责对象显示的代码：

```
public void Show()              // 显示对象
{
    this.visible = true;        //显示该对象，如当选中卡片以后出现的选中标示
}
```

与上面显示对象相对应，下面的代码则实现了对象的隐藏。

```
public void Hide()              // 隐藏对象
{
    this.visible = false;       //隐藏该对象，如隐藏已经被消掉的卡片
```

```
        }
    }
}
```

12.5 卡片类Card源代码详解

```
public class Card : Object
{
```

“：Object”表示Card类是Object类的子类，它继承了父类的所有功能，同时又具备自己的一些特性。下面就来看看Card类中成员变量所代表的含义。

12.5.1 成员变量的含义

Card类对应的是程序中单个的小卡片，在这个类中各个成员变量和常量的含义如表12-2所示。

表12-2 Card类中成员变量和常量的含义

名　称	类　型	含　义
TYPE_MAX	常量const	卡片类型的总数。苹果、梨子、西瓜等都算作不同类型的卡片
WIDTH	常量const	卡片的宽度
HEIGHT	常量const	卡片的高度
type	变量int	卡片的种类。在这里其取值范围为0~12

```
public int Type
{
    get
    {
        return this.type;                    //返回所属卡片的种类
    }
    set
    {
        this.type = value;
        this.srcX = WIDTH * this.type;  //根据卡片的种类，在Card.bmp中选择不同的图片
    }
}
```

上面这段代码的作用为设置返回卡片的类型。其代码表现形式被称为属性。属性是C#语言中独具特色的东西，通过属性可以读写类中的域，同时它还具有一定的保护功能。关于属性的详细讲解，可参考本书前面相关章节的介绍。

12.5.2 关于卡片类的构造函数

```
public Card(Game game, Image image, ImageAttributes imageAttr, int left, int top)
    : base(game, image, imageAttr)
{
```

这个构造函数参数列表后面的“：base（game，image，imageAttr)”表示构造函数在初始化的时候，会将Card参数列表里面的game、image和imageAttr传递给基类构造函数Object（Game

game，Image image，ImageAttributes imageAttr)，这样就实现了初始化所属游戏game、源图像image和颜色属性imageAttr的功能。

而下面这段代码则实现了对象位置的设置、对象大小的设置和对象自身的设置等功能。

```
    Move(left, top);                                    // 设置对象位置
    Resize(WIDTH, HEIGHT);                              // 设置对象大小
    Reset();                                            // 设置对象
}
```

12.5.3 卡片类中其余函数代码的解释

```
    public override void Reset()                        // 重置对象
        {
            Show();                                     // 重置对象为可见
        }
    }
}
```

12.6 卡片集合类Cards源代码详解

这里的Cards类代表卡片的集合，其功能是处理卡片集合相关的事件。用这个集合类中的每一个元素表示每一张卡片，它实现了卡片的功能处理、绘制和重置三个功能。和前面一样，先来看一下这个类中成员变量和常量所代表的含义。

12.6.1 成员变量的含义

Cards类中各个变量和常量的含义如表12-3所示。

表12-3 Cards类中成员变量和常量的含义

名　称	类　型	含　义
LEFT	常量int	卡片集合左上角的X坐标。需要说明的是，这里的卡片集合指的是放在背景画面上的所有卡片。初始值为50
TOP	常量int	卡片集合左上角的Y坐标。初始值为40
WIDTH	常量int	卡片集合的宽度。初始值为500
HEIGHT	常量int	卡片集合的高度。初始值为350。通过上面的这四个初始值可以知道，本游戏的卡片集合画面是放在从坐标（50，40）开始，宽度为500，高度为350的区域中
game	Game类的一个实例	表示对象所属的实例
rows	变量int	卡片集合的行数
cols	变量int	卡片集合的列数
count	变量int	卡片集合中卡片的数量。在本程序中一共有70张卡片
selected	变量int	表示被选中的卡片的编号。在本程序中，用－1来表示没有卡片被选中时的情况，其取值范围为-1~（count-1）
cards	Card类的数组实例	卡片的对象数组。表示数组中的每一个值都是Card类型的

从前面的介绍中知道，每一个卡片的大小是50 × 50，而卡片区域的宽为500，500 ÷ 50=10，

也就是说每一行都有10张卡片；同理，一共有7行卡片。因此游戏开始的时候，整个背景画面上有70张卡片，因此本游戏中selected的取值范围为-1~69。初始化时卡片的数量可以从图12-2中很清楚地看出来。

从该类的构造函数中一定会发现，我们是通过计算得出count的值，那么为什么在这里没有直接把count的值写成70呢？这是因为当卡片大小发生变化的时候，卡片的数量自然也会发生变化（屏幕上的显示区域固定的时候），写成公式计算的方式就可以避免对此部分程序代码的修改，增强了程序的可维护性。如果相关代码都是按照数字大小来进行编写的，那么就需要修改所有相关的数据信息，不利于程序的扩展。

12.6.2 关于卡片集合类的构造函数

下面介绍Cards类的构造函数：

```
public Cards(Game game, Image imageCard, ImageAttributes imageAttrCard)
    {
        this.game = game;                          // 设置游戏对象
        this.rows = HEIGHT / Card.HEIGHT;          //计算卡片集合的行数
        this.cols = WIDTH / Card.WIDTH;            //计算卡片集合的列数
```

因为Card类是公共（public）类型的，里面的卡片宽度和卡片高度也被设置成public类型，使得其对外界来说是可见的。因此在这里可以用“Card.”的方式直接访问这两个常量。

```
this.count = this.cols * this.rows;            // 计算卡片数
this.cards = new Card[this.count];             // 生成卡片对象数组

for (int row = 0; row < this.rows; row++)      //遍历卡片集合的每一行
{
    for (int col = 0; col < this.cols; col++) //遍历卡片集合的每一列
    {
        int index = this.cols * row + col;     // 计算卡片索引，其取值范围为0~71
        int left = Card.WIDTH * col + LEFT;    // 计算卡片左上角x 坐标
        int top = Card.HEIGHT * row + TOP;     // 计算卡片左上角y 坐标
```

在上面这段代码中的Card.WIDTH * col得出的是当前卡片的左上角距离第一张卡片左上角的距离。再加上后面的LEFT就得到了当前卡片左上角的X坐标。同理，也可以得到当前卡片左上角的Y坐标。

```
int type = (index / 2) % Card.TYPE_MAX;        // 设置卡片类型为两两相同
```

（index/2）实现了卡片类型两两相同，再进行“%Card.TYPE_MAX”的计算则实现了对20种卡片类型的选择。在这个函数执行完以后，数组集合cards中每相邻的两个卡片的图形是一样的。

```
            // 生成卡片
            this.cards[index]=new Card(game, imageCard, imageAttrCard, left, top);
            this.cards[index].Type = type;
        }
    }
    Reset();
}
```

12.6.3 与卡片处理、绘制、重置相关的函数

本节主要介绍的是与卡片处理、绘制、重置等功能相关的函数。首先来看一下Process函数。

```
public void Process()
{
    // 处理卡片对象数组
    for (int i = 0; i < this.count; i++)
    {
        if (this.cards[i].Visible)
        {
            this.cards[i].Process();
        }
    }
}
```

细心的读者一定会发现，上面这段代码中调用的cards[i].Process（）实际上没有实现任何的功能，因为Card类中的Process函数没有实现任何功能，那么为什么还要添加上这段代码呢？这是因为在面向对象的程序设计中，Cards类既然作为一个集合类，就需要处理里面的每一个元素，这是作为集合类的一个要求。因此在这里需要添加上这一段代码，以保证Cards类的完整性。

```
public void Draw(Graphics gx)              // 绘制卡片集合
{
    for (int i = 0; i < this.count; i++)
    {
        if (this.cards[i].Visible)
        {
            this.cards[i].Draw(gx);     //调用Card类中的Draw()函数进行绘制
        }
    }
}
```

而下面这段代码则实现了对卡片集合的重新设置。

```
public void Reset(        )                        // 重置集合
{
    for (int i = 0; i < this.count; i++)       // 重置卡片对象数组
    {
        this.cards[i].Reset();
    }

    for (int i = 1; i < this.count; i++)        // 实现对卡片对象数组的随机排序
    {
        int j = this.game.Random.Next(i, this.count);
        int type = this.cards[i - 1].Type;
        this.cards[i - 1].Type = this.cards[j].Type;
        this.cards[j].Type = type;
    }
```

下面就看看是如何通过for循环语句实现卡片数组的随机排序的。上面循环语句里面的

Random.Next是返回大于等于i,且小于this.count的整数。从前面的介绍中可以知道，当初始化卡片数组的时候，每相邻两个卡片的种类是一样的，在这里只需要打乱它们的排列顺序就可以了。

其随机排序的过程为：

1）通过上面提到的随机函数来得到一个大于等于i，且小于等于this.count的随机数j；

2）把[i-1]中的卡片类型和[j]中的卡片类型进行交换，这样就实现了卡片的随机交换功能。

第1步通过前面卡片数组的初始化实现了卡片类型的配对，而第2步则在配对的基础上实现了卡片类型的随机交换。实际上，在程序设计的过程中，尤其是在游戏程序的设计过程中，随机数的应用是非常广泛的，如程序背景画面的随机更新、背景音乐的随机播放、背景图案的随机抖动等都需要借助随机函数的帮助。

```
        this.selected = -1;          // 重置选中卡片编号
    }
```

在上面的一行代码中，因为所有的卡片被重置以后，没有卡片被选中，因此selected被设置成－1。

12.6.4 游戏程序中对屏幕点击的处理

```
public void Hit(int x, int y)              //处理智能设备屏幕的点击
{
    // 当点击屏幕的位置处于卡片显示区域外面的时候
    if ((y < TOP) || (y >= TOP + HEIGHT) || (x < LEFT) || (x >= LEFT + WIDTH))
    {
        return;                             //不做任何处理
    }

    // 当点击屏幕的位置处于卡片显示范围内的时候，计算鼠标点击处的行和列
    int row = (y - TOP) / Card.HEIGHT;
    int col = (x - LEFT) / Card.WIDTH;

    // 被选中卡片的索引编号
    int selected = this.cols * row + col;

    if (this.cards[selected].Visible)  //当选中的卡片是可见的时候
    {
        if (this.selected == -1)        // 当点击位置上的卡片还没有被选中的时候
        {
            this.selected = selected;   // 选中该卡片
```

下面解释一下this.selected=selected这个语句。this修饰的是Cards类中的成员变量selected，等号后面出现的则是Hit函数中的局部变量selected。两者尽管名称是一样的，但是所代表的含义却截然不同，这一点请注意。

```
else if (selected == this.selected)     // 当再次选中同一张卡片的时候
{
    //不做任何处理
```

```
}
```

在上面的if语句中没有做任何处理，对应程序中的表示效果就是，即使两次点击同一张卡片，卡片仍旧处于被选中的状态。如果想在第2次点击的时候取消选中该卡片，则只需要在这个if语句的结构体中添加this.selected=-1即可。

```
else if (!Linked(this.selected, selected))
{
    this.selected = selected;              // 把选中卡片的编号进行更新
}
```

上面这个if语句对应的情况是，当选中的两张卡片不可连通的时候，而下面的else语言自然就对应当选中的两张卡片可以连通的情况。

```
    else
    {
        this.cards[this.selected].Hide();       //消掉第一次选中的卡片
        this.cards[selected].Hide();            //消掉第二次选中的卡片
        this.selected = -1;
        }
    }
}
```

在这里之所以把this.selected = −1，是因为当两张相同的卡片消掉以后，屏幕上就没有卡片被选中，因此需要被设置为−1。

12.6.5 判断两张卡片之间是否可以连通

接下来，就看看在程序代码中如何实现判断两张卡片是否连通。

根据游戏的规则，每两张相同卡片之间的拐点不能超过两个，并且拐点之间不能有别的方块进行阻挡。因此要判断两张卡片是否可以连通，还需要判断中间的两个拐点和其余部分是否可以连通。图12-9已经清晰地标明了起始点、目标点、拐点1和拐点2。而要想判断这4个点是否连通，则需要判断每两个点之间是否可以连通。基于这样的一种判断思路，分别设计了Linked（int src, int dest）、Connected（int row1, int col1, int row2, int col2, int row3, int col3, int row4, int col4）、Through（int srcRow, int srcCol, int destRow, int deskCol, bool skipSrc）3个函数。用（row1, col1）代表起始点所处的行列；（row4, col4）代表目标点所处的行列；（row2, col2）和（row3, col3）代表两个拐点所处的行和列。

接下来需要考虑的问题就是如何选定中间的两个拐点。采取的方法为：

1）选定起始点和目标点两者之间的中心点。

2）以中心点所处的行为基线，向下寻找两个拐点。其中拐点1所处的列等于起始点的列，拐点2所处的列等于目标点的列。

3）以中心点所处的列为基线，向左寻找两个拐点。其中拐点1所处的行等于起始点的行，拐点2所处的行等于目标点的行。

4）以中心点所处的行为基线，向上寻找两个拐点。其中拐点1所处的列等于起始点的列，拐点2所处的列等于目标点的列。

5）以中心点所处的列为基线，向右寻找两个拐点。其中拐点1所处的行等于起始点的行，拐

点2所处的行等于目标点的行。

6）重复步骤2～5，直到找到合适的两个拐点为止。

这一过程如图12-10所示。其中“对号”代表起始点；“禁止符号”代表目标点；“叉号”代表中心点；数字代表寻找拐点的4个方向；实心圆代表拐点1；空心圆代表拐点2。

图12-9　判断4个点是否连通

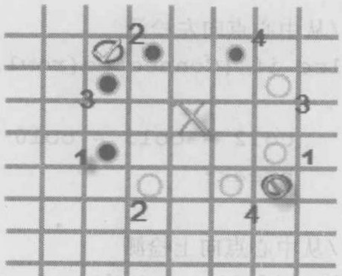


图12-10　寻找两个合适的拐点

看完了上面关于算法的介绍以后，下面就来看一下具体代码的含义。

```
    public bool Linked(int src, int dest)        // 返回两张卡片是否可以连通
{
if (this.cards[src].Type != this.cards[dest].Type)
{
    return false;
}
```

上面这段代码中的if语句对应的情况是：卡片类型不相同的时候，自然两张卡片之间不能连通，因此直接返回即可。

```
int row1 = src / cols;                    //起始卡片的行
int col1 = src % cols;                    //起始卡片的列

int row4 = dest / cols;                   //目标卡片的行
int col4 = dest % cols;                   //目标卡片的列

int row2 = row1;                          // 初始化拐点1的行
int col2 = col1;                          // 初始化拐点1的列

int row3 = row4;                          // 初始化拐点2的行
int col3 = col4;                          // 初始化拐点2的列

int row0 = (row1 + row4) / 2;             // 中心点所处的行
int col0 = (col1 + col4) / 2;             // 中心点所处的列

bool inside = true;                       //是否所有可能的通路都在边界内或边界上
```

下面的代码实现了从中心点向外扩散判断的检测算法。

```
int i = 0;
do
```

```
{
    //从中心点向下检测
    if (Connected(row1, col1, row0 - i, col2, row0 - i, col3, row4, col4))
    {
        row2 = row3 = row0 - i;
        break;
    }
    //从中心点向左检测
    else if (Connected(row1, col1, row2, col0 - i, row3, col0 - i, row4, col4))
    {
        col2 = col3 = col0 - i;
        break;
    }
    //从中心点向上检测
    else if (Connected(row1, col1, row0 + i + 1, col2, row0 + i + 1, col3, row4, col4))
    {
        row2 = row3 = row0 + i + 1;
        break;
    }
    //从中心点向右检测
    else if (Connected(row1, col1, row2, col0 + i + 1, row3, col0 + i + 1, row4, col4))
    {
        col2 = col3 = col0 + i + 1;
        break;
    }
    i++;
    inside = (row0 - i >= -1) || (row0 + i + 1 <= this.rows) ||
        (col0 - i >= -1) || (col0 + i + 1 <= this.cols);
}
while (inside);
```

在这里特别注意一下inside的作用。inside除了具备使循环退出的功能以外，还可以避免程序的死循环。当去掉上面4个或条件的判断语句以后，如果点击两个无法连通的相同类型的图片，程序就会陷入到死循环中。这是因为i的值会一直增加，而无法找到一个退出的条件。此时，尽管模拟器中的鼠标可以移动，但是程序已经“卡”在这里，无法调用其余的函数了。感兴趣的读者不妨测试一下，看看会有什么现象发生。

在这里之所以采用了4个“或”条件判断语句，是为了保证只要有一个方向处在控制范围内(图片显示的区域加上紧邻的一圈，因为最外侧图片连通的时候，需要紧邻一圈的判断运算)，程序就可以继续检测其余的方向。

再来看看下面这段代码：

```
if (inside)                                    // 当连通的道路在控制范围内的时候
{
    // 绘制连接线
    this.game.DrawLines(
        LEFT + col1 * Card.WIDTH + Card.WIDTH / 2,
```

```
        TOP + row1 * Card.HEIGHT + Card.HEIGHT / 2,
        LEFT + col2 * Card.WIDTH + Card.WIDTH / 2,
        TOP + row2 * Card.HEIGHT + Card.HEIGHT / 2,
        LEFT + col3 * Card.WIDTH + Card.WIDTH / 2,
        TOP + row3 * Card.HEIGHT + Card.HEIGHT / 2,
        LEFT + col4 * Card.WIDTH + Card.WIDTH / 2,
        TOP + row4 * Card.HEIGHT + Card.HEIGHT / 2);
    }
    return inside;
}
```

上面的代码涉及DrawLines，这个函数是在后面要介绍的Game类中定义的，关于这个函数的具体讲解，可参考本章后面的相关内容。

下面这段代码的作用是判断起始点、目标点和两个拐点之间是否依次连通。只有各个点依次连通以后才返回真值。

```
public bool Connected(int row1, int col1, int row2, int col2, int row3, int
col3, int row4, int col4)
{
    return Through(row1, col1, row2, col2, true) &&
        Through(row2, col2, row3, col3, (row2 == row1) && (col2 == col1)) &&
        Through(row3, col3, row4, col4,);
}
```

上面一段代码中的row1和col1分别代表起始格所处的行和列；row4和col4代表目标格的行和列；中间的4个参数则代表拐点的行和列。

下面一段代码的作用则是判断两个格之间是否可以连通，并且之间没有障碍。其中srcRow和srcCol分别代表起始格的行和列；destRow和destCol则代表目标格的行和列。参数skipSrc则表示是否判断起始格为障碍物。

在判断每两张卡片之间是否连通的时候，都是把起始卡片作为初始点，结束卡片的"前一张"作为终止点，来判断两者之间是否连通的。但是当起始点和终止点为同一个点的时候，按照上面提到的算法则会出现判断错误，因此在这里引入了skipSrc变量，其作用为是否判断起始点。因此在Connected函数中，判断中间两个拐点之间以及第2个拐点和目标点之间是否连通的时候，需要用(row2 == row1) && (col2 == col1)和(row3 == row1) && (col3 == col1)的组合判断。因为当同时满足上面的与运算条件的时候，表示起始点和目标点为同一个点，这样其值就为true。也就表示可以直接跳过开始点了。

通过上面这种方法，就可以有效地解决判断错误的问题。此判断过程如图12-11所示。

图12-11　判断4格之间是否连通

下面这段代码实现的功能是判断两个格子之间是否可以连通。

```
public bool Through(int srcRow, int srcCol, int destRow, int destCol, bool skipSrc)
{
    if ((srcRow == destRow) && (srcCol == destCol))   // 同一格
    {
```

```
        return true;
    }
    else if (srcRow == destRow)                        // 在同一行
    {
        int row = srcRow;
        if ((row < -1) || (row > this.rows))
        {
            return false;                              // 不能通过
        }
```

上面这个if语言对应的情况是当行处在卡片区域外扩1圈外侧的时候。此时自然处于无法通过的状态。因此直接返回false。

```
else if ((row == -1) || (row == this.rows))
{
    return true;
}
```

上面这个else if语句对应的情况是当行在卡片区域外扩1圈边界上的时候。在这种情况下，外扩的1圈一定没有卡片阻挡，所以连接线是一定可以通过的。因此在这里直接返回真值即可。

```
else                                   // 行在边界内
{
    int step = Math.Sign(destCol - srcCol);
```

- 当（destcol-srcCol）大于0的时候，Math.Sign（）返回1；
- 当（destcol-srcCol）等于0的时候，Math.Sign（）返回0；
- 当（destcol-srcCol）小于0的时候，Math.Sign（）返回-1；

```
for (int col = srcCol + (skipSrc ? step : 0); col != destCol; col += step)
{
```

上面代码中的（skipSrc ? step : 0）语句的含义为：当skipSrc为真的时候，取step的值作为结果；当skipSrc为假的时候，取0作为结果。

在判断同一行卡片的时候，根据是否需要跳过起始卡片来决定循环判断时列的起始数，其循环终止条件为判断到目标格的前一个位置为止。可以参照图12-10来理解上面描述的过程。

```
            if ((col < -1) || (col > this.cols) ||             // 列在边界外
                ((col >= 0) && (col < this.cols) &&            // 列在边界内
                this.cards[this.cols * row + col].Visible))
            {
                return false;
            }
        }
        return true;
    }
}
```

上面这个if语句所对应的情况是当列在边界外，或者列在边界内，同时两者之间有障碍的时候。此时直接返回false即可。

和对同一行的判断方法一样，下面的代码实现了对同一列的判断。

```
        else if (srcCol == destCol)                          // 在同一列
        {
            int col = srcCol;
            if ((col < -1) || (col > cols))                  // 列在边界外
            {
                return false;                                // 不能通过
            }
            else if ((col == -1) || (col == cols))           // 列在边界上
            {
                return true;                                 // 能通过
            }
            else // 列在边界内
            {
                int step = Math.Sign(destRow - srcRow);
                for (int row= srcRow + (skipSrc ? step : 0); row !=
                destRow; row += step)
                {
                    if ((row < -1) || (row > this.rows) ||     // 行在边界外
                        ((row >= 0) && (row < this.rows) &&    // 或行在边界内
                        this.cards[this.cols * row + col].Visible))
                                                               // 但有障碍
                    {
                        return false;
                    }
                }
                return true;
            }
        }
        else // 不在同一条线上
        {
            return false;
        }
    }
}
}
```

学习完了Cards类，接下来就看看Program类的作用。

12.7 游戏类Program源代码详解

12.7.1 成员变量的含义

游戏（Game）类中各个成员变量的含义如表12-4所示。

表12-4 Game类中成员变量的含义

名　称	含　义
gxPhys	在前台绘制图形，用于图形的显示
imageOff	在后台需要绘制的图像
gxOff	在后台绘制图形，用于图形的显示
imageBack	背景源图像
imageAttr	指定源图像的透明颜色信息。和前面一样，本游戏中统一为Color.FromArgb(255,0,255)
cards	卡片集合对象
random	随机数生成器
running	游戏是否正在运行

在上面对成员变量的介绍中，一定会注意到提到的后台绘制图像和前台绘制图形。这到底有什么作用呢？下面就来看一下。

在游戏程序设计中，如果把所有的游戏素材直接绘制到屏幕上，就会出现由于绘制的图像资源过多和图像资源更新过快而导致的屏幕闪烁现象。这会严重地影响玩家的游戏感受。因此，在这里采取了一种被称为“离屏技术”的画面绘制方法。具体的设计思路是：首先创建一个虚拟的屏幕，然后把将要绘制的图形画面绘制到这个虚拟屏幕上，最后把虚拟屏幕上的所有内容复制到可以显示的真正的物理屏幕上。使用这种技术可以很好地避免屏幕的闪烁现象。这是因为在绘制到虚拟屏幕的时候，真正的屏幕是不发生变化的。同时，在将整个虚拟屏幕上的内容复制到真正的物理屏幕的时候，所花费的时间很短。在内容变化不大的时候，几乎看不到任何的闪烁。

离屏技术主要有下面3个步骤：

1）创建用于离屏显示的虚拟屏幕；

2）在创建的虚拟屏幕上绘制画面；

3）将虚拟屏幕上的整个画面复制到真正的物理屏幕上。

在这个游戏中，gxPhys对应的就是物理屏幕，gxOff对应的就是后台的虚拟屏幕，这就从编码上实现了离屏显示技术。

12.7.2 游戏程序的初始化

```
public Game()
{
    Init();                          //初始化游戏
    MainLoop();                      //进入到游戏的主循环中
}
```

为了使得在每一次游戏开始的时候，卡片的排列顺序都不相同，在程序代码中就需要用随机函数生成不同的随机数。下面的一段代码就实现了上述功能。

```
public Random Random                 //随机函数
{
    get
    {
        return this.random;
    }
```

```
}
```

下面的一段代码则实现对应用程序的初始化，

```
protected void Init()                          // 初始化应用程序
{
    this.gxPhys = this.CreateGraphics();       // 在前台物理屏上绘制图像

    // 在后台需要绘制的图像，就是前面提到的图片资源中的Background.bmp文件
    this.imageOff = Resources.Background;
    // 后台绘图面
    this.gxOff = Graphics.FromImage(this.imageOff);
```

上面代码中FromImage的作用为用ImageeOff创建新的gxOff对象。而下面这段代码则实现了设备背景源图像、指定源图像的透明色信息的功能。

```
// 背景源图像，同样为图片资源中的Background.bmp文件
this.imageBack = Resources.Background;

// 指定源图像的透明色信息
this.imageAttr = new ImageAttributes();
this.imageAttr.SetColorKey(Color.FromArgb(255, 0, 255), Color.
FromArgb(255, 0, 255));
```

上面语句中SetColorKey的作用是为两个颜色值之间的颜色设置颜色属性。其函数原型为：

```
SetColorKey (Color colorLow, Color colorHigh)
```

其中第一个颜色属性表示低色键值，第二个颜色属性表示高色键值。对于任何颜色，只要它的三种颜色成分（红、绿、蓝）都处于高低色键的对应成分之间，它的颜色就会成为透明色。

下面一行代码的作用是生成卡片集合。

```
this.cards = new Cards(this, Resources.Card, this.imageAttr);
```

下面一行代码则实现了对窗体标题的设定。可以非常清晰地看到，在这里把窗体的标题设置成了Link Game。

```
this.Text = "Link Game";
```

而下面代码的作用则为显示窗体。

```
    this.Show();
}
```

下面的MainLoop函数则为该游戏程序的主循环函数。

```
protected void MainLoop()                    // 游戏主循环
{
    while (this.running)                     // 当游戏处于运行状态的时候
    {
        // 让系统能处理其他应用程序的事件，避免它们无响应
        Application.DoEvents();
```

Application.DoEvents()的作用是处理在消息队列中所有的Windows消息，如点击屏幕等。如果去掉这个函数，程序将无法响应按键处理。感兴趣的读者可以尝试一下去掉这行代码，看看会有什么现象发生。

```
        ProcessAll();                   // 处理所有游戏对象
        DrawAll();                      // 绘制所有游戏对象
    }
    Close();                            // 关闭窗体
}
```

12.7.3 游戏对象的处理

下面就来看一下与游戏对象处理相关的函数。首先介绍的就是处理所有游戏对象的ProcessAll函数。

```
public void ProcessAll()
{
    this.cards.Process();               // 处理卡片集合
}
```

下面的DrawAll函数实现了对所有游戏对象的绘制。

```
public void DrawAll()                   // 绘制所有游戏对象
{
    // 绘制背景，以覆盖并准备重绘前景
    this.gxOff.DrawImage(this.imageBack, 0, 0);

    // 绘制卡片集合，调用了Cards类中的Draw函数
    this.cards.Draw(this.gxOff);

    // 刷新前台显示屏幕，此时将后台画面一次性绘制到前台，离屏技术的体现
    this.gxPhys.DrawImage(this.imageOff, 0, 0);
}
```

函数DrawImage的作用是在指定位置绘制指定的Image对象。在本程序中，就是从显示屏幕的最左上角开始绘制整个背景画面。

下面的ResetAll函数实现对所有游戏对象的重置。

```
public void ResetAll()                  // 重置所有游戏对象
{
    this.cards.Reset();                 // 重置卡片集合
}
```

下面DrawLines函数的作用是绘制卡片之间的连接线。

```
public void DrawLines(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4)
{
    Pen pen = new Pen(Color.Red);       //设置画笔的颜色为红色
```

下面的三行代码实现了在起始点、目标点和两个拐点之间绘制连接线。

```
    this.gxPhys.DrawLine(pen, x1, y1, x2, y2);
    this.gxPhys.DrawLine(pen, x2, y2, x3, y3);
    this.gxPhys.DrawLine(pen, x3, y3, x4, y4);
}
```

下面来看一下DrawLine函数在C#中的函数原型：

```
Public void DrawLine(
    Pen pen,
    int x1,
    int y1,
    int x2,
    int y2
);
```

其中pen对象表示线条的颜色、宽度或者样式。x1为第一个点的x坐标，y1为第一个点的y坐标；x2为第二个点的x坐标，y2为第二个点的y坐标。

12.7.4 游戏程序的主函数

Main函数为游戏程序的主函数，启动应用程序时使用。

```
public static void Main()
{
    Application.Run(new Game());        // 生成游戏窗体，开始游戏
}
```

而下面的OnMouseDown函数则实现了对点击屏幕的处理。

```
        protected override void OnMouseDown(MouseEventArgs mouseEventArgs)
        {
            if ((mouseEventArgs.Y < 28) && (mouseEventArgs.X >= 212))
                                                    // 点击关闭按钮区域
            {
                this.running = false;               // 结束游戏
            }
            else // 点击其他地方
            {
                this.cards.Hit(mouseEventArgs.X, mouseEventArgs.Y);
                                                    // 让卡片集合处理本次点击
            }
        }
    }
}
```

使用override来修饰函数OnMouseDown，表示对OnMouseDown方法进行了重写。

第13章　使用Visual Basic设计Windows Embedded CE 6.0版俄罗斯方块

13.1　程序构建过程

本章将介绍如何使用Visual Basic设计Windows Embedded CE 6.0版俄罗斯方块的过程。首先来看一下这个程序是如何构建的。

首先启动Visual Studio 2005，单击菜单中的File→New→Project，此时会出现图13-1所示的画面。

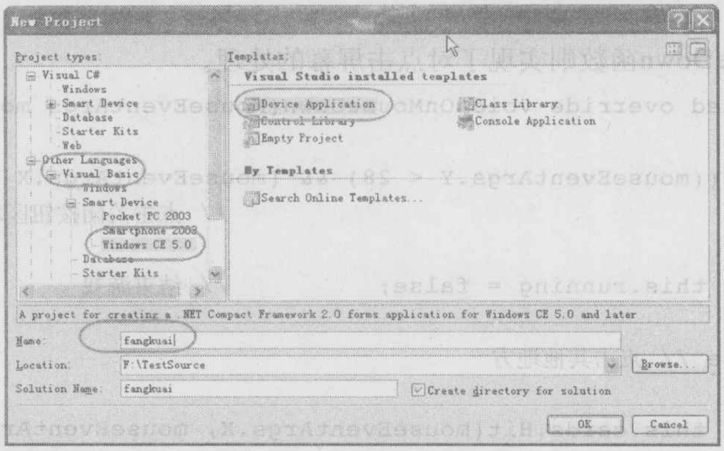


图13-1　程序构建步骤1

在这个画面中，需要进行如下一些选择和设置：

- 在Other Languages中选择Visual Basic。
- 在Smart Device中选择Windows CE 6.0。
- 在Visual Studio installed templates中选择Device Application。
- 在Name中输入fangkuai作为工程名称。

设置好以后，单击OK按钮，进入到图13-2所示的画面。从这个画面中可以看到，Visual Studio 2005自动生成了一个图形编辑界面，我们可以通过拖拽相应的控件或编写相应的代码来进行程序的用户界面设计工作。

在设置好程序框架以后，接下来要做的就是添加相应的控件。首先把鼠标放在Visual Studio 2005里面的Toolbox上，此时会自动展现出控件工具栏。图13-3展示了Visual Studio 2005里面的工具栏。

在Visual Basic版“俄罗斯方块”程序中，用到了下列控件：

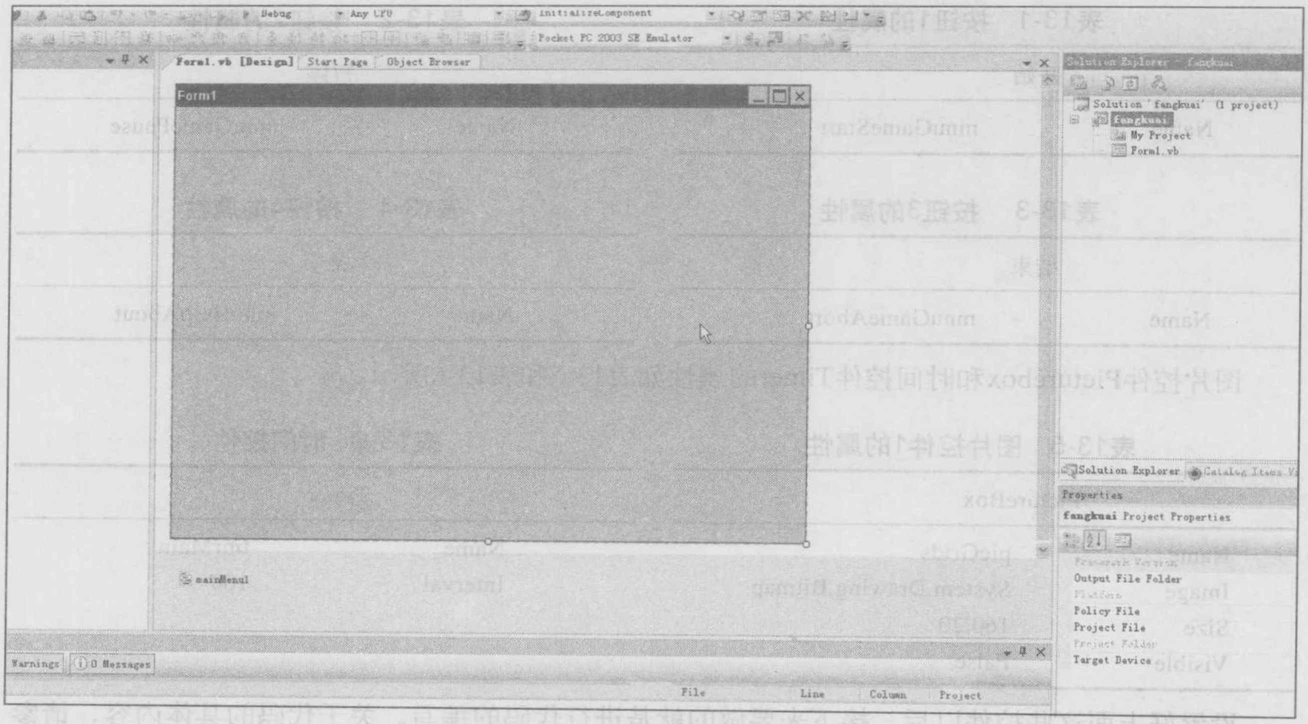


图13-2 程序设计画面

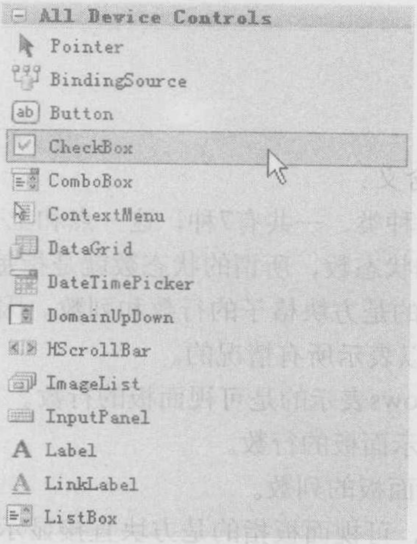


图13-3 Visual Studio 2005中的工具栏

- 时间控件Timer。
- 菜单控件MainMenu。
- 图片控件PictureBox。

通过菜单控件MainMenu，可以很方便地设计出程序所需要的菜单选项。在这里，一共设置了4个菜单，分别为begin、pause、stop和about。各个菜单的属性设置参见表13-1～表13-4。

表13-1 按钮1的属性

开始	
Name	mnuGameStart

表13-2 按钮2的属性

暂停	
Name	mnuGamePause

表13-3 按钮3的属性

结束	
Name	mnuGameAbort

表13-4 按钮4的属性

关于	
Name	mnuHelpAbout

图片控件Picturebox和时间控件Timer的属性如表13-5和表13-6所示。

表13-5 图片控件1的属性

PictureBox	
Name	picGrids
Image	System.Drawing.Bitmap
Size	160,20
Visible	False

表13-6 时间控件

Timer	
Name	tmrMain
Interval	100

设置好上面这些控件以后，接下来要做的就是进行代码的编写。关于代码的具体内容，请参考本书的附带光盘。

13.2 源代码详解

13.2.1 全局量的声明

首先看一下全局常量的一些含义：

- PieceTypes 表示的是方块的种类，一共有7种，这一点和上一章的表示一样。
- PieceStates 表示的是方块的状态数，所谓的状态数就是指每一种方块的四个方向。
- PieceRows和PieceCols表示的是方块格子的行数和列数。因为即使是最长的“一”字型方块也只有4个格子，因此是可以表示所有情况的。
- VisBoardRows：VisBoardRows表示的是可视面板的行数。
- BoardRows：BoardRows表示面板的行数。
- BoardCols：BoardCols表示面板的列数。

在这里有一点需要指出的是，可视面板指的是方块直接显示的地方，而面板指的是方块初始化的位置，当方块在这个位置的时候是不显示的。当其开始下落的时候才会在可视面板上显示出来。这也就是让BoardRows = VisBoardRows + 4的原因。

下面来看一下全局变量的一些含义。

- Board：Board表示面板，也就是用来放置方块的地方。
- Piece：表示方块格子。里面的参数分别表示方块的种类、方块的方向、方块格子的行值和列值。
- TypeCurr：代表当前方块的类型。

- StateCurr：代表当前方块的状态。
- RowCurr：代表当前方块的行。
- ColCurr：代表当前方块的列。
- TypeNext：代表下一个方块的类型。
- StateNext：代表下一个方块的状态。
- Playing：表示游戏是否处于运行状态。
- Graphics：表示声明一个图形类。

13.2.2　相关按键和菜单的处理

图13-4展示了设计好的菜单样式。

那么如何通过程序代码来实现对菜单按钮的事件响应呢？下面就来看一下。

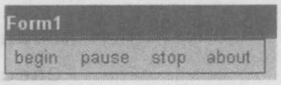


图13-4　设计好的菜单样式

```
Private Sub mnuGameStart_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuGameStart.Click
        StartGame()
    End Sub
```

上面一段代码是处理单击“开始”菜单的时候发生的事件。从里面的StartGame()可以看出，此时游戏开始。

```
Private Sub mnuGamePause_Click(ByVal sender As System.Object, ByVal e As System.
EventArgs) Handles mnuGamePause.Click
    PauseGame()
End Sub
```

上面一段代码是处理单击“暂停”菜单的时候发生的事件。从里面的PauseGame()可以看出，此时游戏进入到暂停状态。

```
Private Sub mnuGameAbort_Click(ByVal sender As System.Object, ByVal e As System.
EventArgs) Handles mnuGameAbort.Click
    AbortGame()
End Sub
```

上面一段代码是处理单击“结束”菜单的时候发生的事件。此时游戏停止。

下面的一段代码实现了按键的处理。

```
Private Sub Form1_KeyDown(ByVal sender As Object, ByVal e As System.Windows.
Forms.KeyEventArgs) Handles MyBase.KeyDown
    If Playing Then                                     '当游戏处于运行状态的时候

    If e.KeyCode = Keys.Down Then                       '方块下降的时候
        MovePiece(-1, 0)
    ElseIf e.KeyCode = Keys.Left Then                   '方块左移的时候
        MovePiece(0, -1)
    ElseIf e.KeyCode = Keys.Right Then                  '方块右移的时候
        MovePiece(0, 1)
    ElseIf e.KeyCode = Keys.Up Then                     '方块变形的时候
        RotatePiece(1)
```

```
    End If
```

13.2.3 方块的下落

```
Private Sub tmrMain_Tick(ByVal sender As System.Object, ByVal e As System.
EventArgs) Handles tmrMain.Tick

    If Not Playing Then Exit Sub

    If Not MovePiece(-1, 0) Then                       '方块不能再下降的时候
        PieceDown()

        If RowCurr < VisBoardRows Then                 '行数小于可视面板行数的时候
            CheckLine()
            LoadPiece()
        Else
            AbortGame()
            MsgBox("真遗憾，你输了！")
        End If
    End If
End Sub
```

13.2.4 游戏的初始化

下面来看一下游戏初始化时候的代码。

```
Private Sub InitGame()
    Playing = False
    If Not LoadGame() Then                             '对异常的处理
        MsgBox("载入数据出错！")                        '弹出相应的消息
    End If
    RestoreGame()                                      '对游戏进行重新设置
End Sub
```

在上面的代码中，添加if语句是为了增强程序的健壮性。当初始化游戏的时候，若无法加载数据文件就会弹出对话框。这样就会防止程序出错。

13.2.5 游戏的重置

```
Private Sub RestoreGame()
    Dim r, c As Integer
    For r = 0 To BoardRows - 1
        For c = 0 To BoardCols - 1
            Board(r, c) = 0                            '初始化
            DrawGrid(r, c)                             '绘制黑色的背景
        Next
    Next
    LoadPiece()
End Sub
```

13.2.6 游戏开始

在游戏开始的时候，需要重新设置游戏。这也就是调用RestoreGame的原因。

```
Private Sub StartGame()
    RestoreGame()
    Playing = True
End Sub
```

13.2.7 游戏的暂停和退出

```
Private Sub PauseGame()
    Playing = Not Playing        '游戏暂停的时候，游戏状态设置为非运行
End Sub

Private Sub AbortGame()          '游戏退出
    Playing = False
    RestoreGame()
End Sub
```

13.2.8 游戏数据的加载

在这个游戏中，用1表示方块格子中有方块，0表示方块格子中没有方块。这样，就把各种类型的方块用代码很好地表现了出来。从下面的代码表示中，可以很容易地看出各种方块的类型。

需要说明的是，在这里即使同一种方块，在不同的旋转状态下也采用了代码上有所区别的表示。

由于此段代码相对简单，这里只列出一部分，完整内容可参照前面代码部分的介绍。

```
    Private Function LoadGame() As Boolean
        Dim t As Integer, s As Integer, r As Integer, c As Integer
        Dim p(,,,) As Integer = { _
{ _
{ _
{0, 1, 0, 0}, _
{0, 1, 0, 0}, _
{0, 1, 0, 0}, _
{0, 1, 0, 0}}, _
{ _
{0, 0, 0, 0}, _
{1, 1, 1, 1}, _
{0, 0, 0, 0}, _
{0, 0, 0, 0}}, _
{ _
{0, 1, 0, 0}, _
{0, 1, 0, 0}, _
{0, 1, 0, 0}, _
{0, 1, 0, 0}}, _
{ _
```

```
{0, 0, 0, 0}, _
{1, 1, 1, 1}, _
{0, 0, 0, 0}, _
{0, 0, 0, 0}}}, _
{ _
{ _
{0, 1, 1, 0}, _
{0, 0, 1, 0}, _
{0, 0, 1, 0}, _
{0, 0, 0, 0}}, _
{ _
{0, 0, 0, 0}, _
{0, 0, 1, 0}, _
{1, 1, 1, 0}, _
{0, 0, 0, 0}}, _
......
```

13.2.9 方块的加载

下面的一段代码实现了不同方块的加载。

```
Private Sub LoadPiece()

    TypeCurr = TypeNext
    StateCurr = StateNext
    ColCurr = (BoardCols - PieceCols) \ 2              '当前方块所处列的中心点
    RowCurr = VisBoardRows                             '当前方块的位置

    TypeNext = Int(Rnd() * PieceTypes)                 '下一个方块的类型
    StateNext = Int(Rnd() * PieceStates)               '下一个方块所处的状态

    DrawNextPiece()                                    '绘制下一个方块
End Sub
```

13.2.10 方块的放置

方块下落的时候，需要不断地判断一个位置是否可以放置方块，这个功能的实现就是依靠PiecePutable函数来完成的。下面就介绍这个函数中关键代码的含义。

```
Private Function PiecePutable(ByVal Type As Integer, ByVal State As Integer, _
ByVal Row As Integer, ByVal Col As Integer) As Boolean

    Dim r, c As Integer

    For r = 0 To PieceRows - 1
        For c = 0 To PieceCols - 1
            If Piece(Type, State, r, c) > 0 Then       '当某一点有小方块的时候
                If Row + r < 0 Or Col + c < 0 Or Col + c >= BoardCols Then
                    Return False
```

上面一段代码对应情况分别为方块超出显示区域的上边界，超出左侧边界和超出右侧边界。这种情况下，自然就需要返回false，表示不能移动到那个地方。

再看下面的代码：

```
ElseIf Board(Row + r, Col + c) > 0 Then
```

上面的代码表示当面板上有方块的时候，返回false。

```
                        Return False
                    End If
                End If
            Next
        Next
        Return True
    End Function
```

在经过上面的判断以后，如果一个地方可以放置方块，就需要使用下面的PutPiece函数在相应的位置上放置方块。

```
    Private Function PutPiece(ByVal StateNew As Integer, _
  ByVal RowNew As Integer, ByVal ColNew As Integer) As Boolean

        Dim r, c As Integer
        If PiecePutable(TypeCurr, StateNew, RowNew, ColNew) Then
            ErasePiece(TypeCurr, StateCurr, RowCurr, ColCurr)
            DrawPiece(TypeCurr, StateNew, RowNew, ColNew)
            StateCurr = StateNew
            RowCurr = RowNew
            ColCurr = ColNew
            Return True
        Else
            Return False
        End If
    End Function
```

当判断一个位置可以放置方块的时候，需要按照下面的步骤进行操作：

1）删除当前位置的方块。

2）在新的位置绘制相应的方块。

3）把当前状态标记为新的状态。

4）把当前所处的行标记为新的行。

5）把当前所处的列标记为新的列。

13.2.11 方块的旋转

下面的函数实现了方块旋转变化。

```
Private Function RotatePiece(ByVal StateOff As Integer) As Boolean
    Dim s As Integer
    s = (PieceStates + StateCurr + StateOff) Mod PieceStates
    If Not PutPiece(s, RowCurr, ColCurr) Then
```

```
        If Not PutPiece(s, RowCurr, ColCurr - 1) Then
            If Not PutPiece(s, RowCurr, ColCurr + 1) Then
                If Not PutPiece(s, RowCurr, ColCurr - 2) Then
                    If Not PutPiece(s, RowCurr, ColCurr + 2) Then
                        Return False
                    End If
                End If
            End If
        End If
    End If
    Return True
End Function
```

上面一系列if语句的作用是：

- 当当前位置无法旋转的时候。
- 当试图向左移动1个单位不成功的时候。
- 当试图向右移动1个单位不成功的时候。
- 当试图向左移动2个单位不成功的时候。
- 当试图向右移动2个单位不成功的时候。

13.2.12 方块的移动和落定函数

下面这两段代码分别实现了方块的移动和落定功能。提到方块的移动，自然就会涉及移动的偏移量，这就是在MovePiece函数中采用了RowOff和ColOff两个参数来表示偏移量的原因。

在方块移动函数MovePiece中，还调用了方块落定函数PieceDown。PieceDown函数的作用是对方快的落定进行处理。下面就仔细看一下这两个函数中关键代码的含义。

```
Private Function MovePiece(ByVal RowOff As Integer, ByVal ColOff As Integer) As Boolean
    Return PutPiece(StateCurr, RowCurr + RowOff, ColCurr + ColOff)
End Function
```

Return语句表示方快移动一个（RowOff，ColOff）偏移以后的落定位置。而下面的PieceDown函数中：

```
Private Sub PieceDown()
    Dim r, c As Integer
    For r = 0 To PieceRows - 1                          //遍历所有的行
        For c = 0 To PieceCols - 1                      //遍历所有的列
            If Piece(TypeCurr, StateCurr, r, c) > 0 Then
                Board(RowCurr + r, ColCurr + c) = TypeCurr + 1
            End If
        Next
    Next
End Sub
```

上面这段代码中的if语句表示当方快格子中已经有小方块的时候，此时在面板上记录下方块的类型。下面再来看看在程序代码中是如何实现判断并移除符合消去条件的行的。

13.2.13 判断移除整行

CheckLine函数的作用是判断并且移除整行，下面介绍这段代码中关键代码的含义。

```
Private Sub CheckLine()
    Dim l As Integer
    l = 0
    For r = PieceRows - 1 To 0 Step -1
        If RowCurr + r >= 0 Then
```

上面这个if语句所表示的情况是当处于可视面板范围以内的时候，关于可视面板的定义可参考前面的介绍。

```
For c = 0 To BoardCols - 1
    If Board(RowCurr + r, c) = 0 Then
        Exit For
    End If
Next
```

上面这段代码表示某一行没有被填充满的时候，此时表示当前行不能被溢出，自然就需要退出这个检测过程。

```
            If c >= BoardCols Then
                RemoveLine(RowCurr + r)
                l += 1
            End If
        End If
    Next
```

上面这段代码表示当检测到最右边的时候，移除新添加上的一行。因为当检测到最右侧的时候，表示这一行已经被填充满。

下面这个if语句表示移除一行以后，需要进行得分的设置。按照通常的俄罗斯方块游戏，同时移除的行数越多，分值增加的越多。也就是说，一次移除4行的得分要比每次移除一行，一共移除4次的得分多。这也就是在这里添加上l的原因。

```
    If l > 0 Then SetScore(Score + l * (l + 1) / 2)
End Sub
```

RemoveLine函数的作用是具体实现某一行的移除。其中的参数Row表示移除第Row行。

```
Private Sub RemoveLine(ByVal Row As Integer)
    Dim r As Integer, c As Integer

    For r = Row To BoardRows - 2
        For c = 0 To BoardCols - 1
            Board(r, c) = Board(r + 1, c)
```

上面Board(r, c) = Board(r + 1, c)这一行的代码表示的含义是下一行的方块被替换成上一行的方块。进行完替换以后，就需要在当前位置上绘制出相应的图像。这就是下面一行代码DrawGrid(r, c)的作用。

```
            DrawGrid(r, c)
        Next
```

```
    Next
End Sub
```

13.2.14 程序画面的实现

下面的DrawGrid函数的作用是显示方块，也就是使用这个函数在左边方块的可视面积内的(Row,Col）位置上绘制某种类型的方块。

```
Private Function DrawGrid(ByVal Row As Integer, ByVal Col As Integer)
    DrawLeft(Row, Col, Board(Row, Col))      //在相应位置上绘制方块
End Function
```

在这个程序中，有绘制方块，自然一定会有擦除方块。当某一行被小块填充满以后，就会被消去。此时，就需要ErasePiece函数来擦除相应位置上的方块。看看下面的代码实现：

```
Private Sub ErasePiece(ByVal Type As Integer, ByVal State As Integer,
ByVal Row As Integer, ByVal Col As Integer)
    Dim r, c As Integer
    For r = 0 To PieceRows - 1
        For c = 0 To PieceCols - 1
            If Piece(Type, State, r, c) > 0 Then
                DrawLeft(Row + r, Col + c, 0)
```

上面的Piece(Type, State, r, c) > 0表示当某一位置的方块格子上有小方块的时候，就需要调用下面的DrawLeft函数在（Row+r，Col+c）的位置上绘制黑色背景色。通过这种方式，就实现了方块的擦除。

```
                End If
            Next
        Next
End Sub
```

下面的函数DrawPiece实现了对方块的绘制。

```
Private Sub DrawPiece(ByVal Type As Integer, ByVal State As Integer, ByVal Row
As Integer, ByVal Col As Integer)
    Dim r, c As Integer
    For r = 0 To PieceRows - 1
        For c = 0 To PieceCols - 1
            If Piece(Type, State, r, c) > 0 Then
                DrawLeft(Row + r, Col + c, Type + 1)
```

上面的if语句中，Piece(Type, State, r, c) > 0表示的情况是当某一个方块格子中有小方块的时候，就需要调用DrawLeft函数在左边的显示区域的（Row + r, Col + c, Type + 1)位置上绘制出该方块。

```
            End If
        Next
    Next
End Sub
```

从下面的函数DrawNextPiece的名称中就可以猜到，这个DrawNextPiece函数的作用是在右侧

显示下一个方块类型的地方绘制出下一个将要出现的方块。

```
Private Sub DrawNextPiece()
        Dim r, c As Integer
        For r = 0 To PieceRows - 1
            For c = 0 To PieceCols - 1
                If Piece(TypeNext, StateNext, r, c) > 0 Then
                    DrawRight(r, c, TypeNext + 1)
                Else
                    DrawRight(r, c, 0)
                End If
```

上面这个if语句中对应的情况是当方块格子中有方块的时候，就需要调用DrawRight函数在相应的位置上绘制出下一个方块；当方块格子中没有方块的时候，则会调用DrawRight函数在右侧的面板上绘制出黑色的背景图像。

```
        Next
    Next
End Sub
```

下面的两个函数DrawLeft和DrawRight的作用是在（Row，Col）位置上绘制出相应的方块。其中DrawLeft是在左侧的显示区域中绘制正在下落的方块，DrawRight是在右侧的显示下一个方块的区域中绘制下一个将要出现的方块。

```
Private Sub DrawLeft(ByVal Row As Integer, ByVal Col As Integer, ByVal
GridType As Integer)        ' 在面板(Row, Col) 位置处显示类型为GridType 格的图像
    If Row < VisBoardRows Then
        GraphMain.DrawImage(picGrids.Image, 4 + Col * 16, 4 + (VisBoardRows -
        1 - Row) * 16, _
        New Rectangle(GridType * 20 + 2, 2, 16, 16), GraphicsUnit.Pixel)
    End If
End Sub

Private Sub DrawRight(ByVal Row As Integer, ByVal Col As Integer, ByVal
GridType As Integer)       '在方块预览区(Row, Col)位置处显示类型为GridType格的图像

    GraphMain.DrawImage(picGrids.Image, 260 + Col * 16, 8 + (PieceRows - 1 -
    Row) * 16, _
    New Rectangle(GridType * 20 + 2, 2, 20, 20), GraphicsUnit.Pixel)
End Sub
```

我们注意到，这两个函数涉及DrawImage。就来看看在编译环境中，系统是如何进行提示的，如图13-5所示。

```
▲1 of 4▼   DrawImage (image As System.Drawing.Image, x As Integer, y As Integer)
image: System.Drawing.Image object to draw.
```

图13-5　DrawImage函数的提示信息

从这个系统提示中可以看出，第一个变量代表源图像，第二个和第三个变量分别代表目的位置的横、纵坐标。后面的Rectangle代表源图像的那一部分图像。

下面将以这个代码为例子加以说明。例如：

```
GraphMain.DrawImage(picGrids.Image, 4 + Col * 16, 4 + (VisBoardRows - 1 - Row) * 16, _
      New Rectangle(GridType * 20 + 2, 2, 16, 16), GraphicsUnit.Pixel)
```

上面这个DrawImage函数的含义是：

- 源图像是picGrid，就是那个包含图片的PictureBox控件。
- 目的位置的横坐标是4 + Col * 16。
- 目的位置的纵坐标是4 + (VisBoardRows - 1 - Row) * 16。
- 需要绘制PictureBox控件中的，从GridType * 20 + 2, 2开始，长度分别为16，16的图像。
- DrawImage函数后面的GraphicsUnit.Pixel是度量单位。

13.2.15 程序画面

图13-6展示了“俄罗斯方块”程序启动时的画面；图13-7展示了游戏程序运行中的画面。

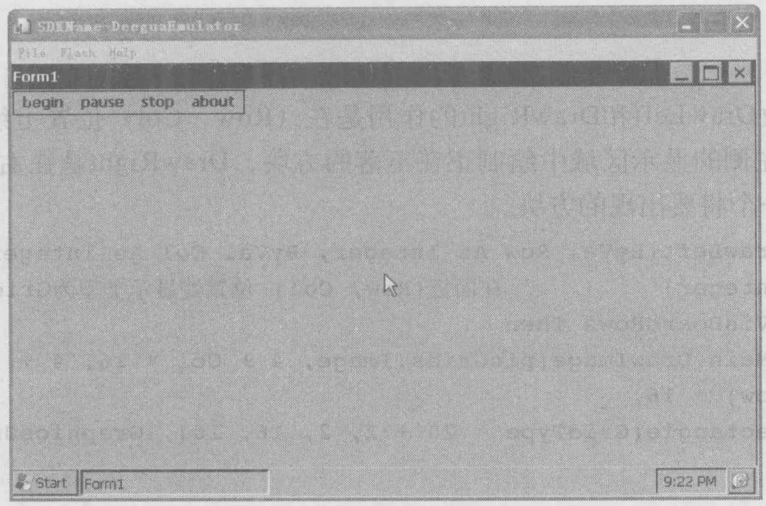


图13-6 程序启动时的画面

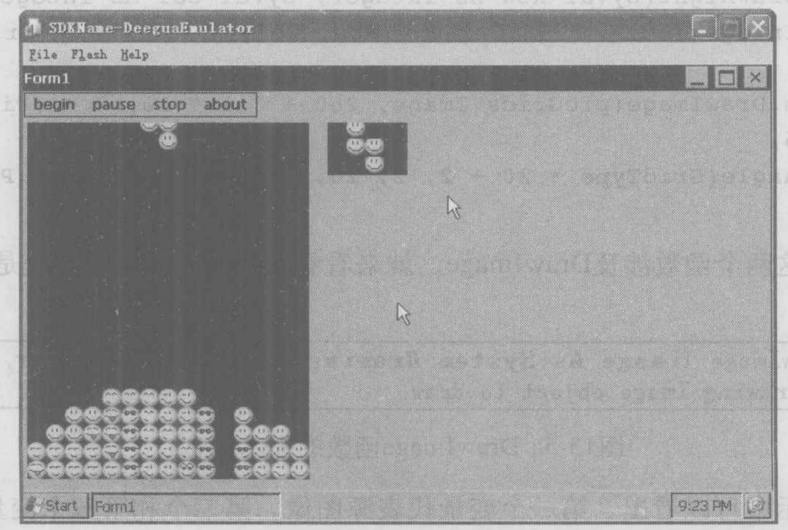


图13-7 程序运行中的画面

第14章 使用Embedded Visual C++进行Windows CE程序设计

在前面几章的内容中，介绍了Windows CE程序设计中一些基本的理论支持。同时，也介绍了采用Embedded Visual C++设计一些简单的Windows CE程序。本章将详细介绍采用Embedded Visual C++进行Windows CE程序设计的基本步骤和基本方法。从中不但可以学习到如何构建一个基于Windows CE的应用程序，还可以了解到整个Windows CE程序实例的开发过程。

在本章中介绍的程序实例为经典的俄罗斯方块游戏程序。在开始学习如何构建这个程序以前，先来了解一下Embedded Visual C++。

14.1 Embedded Visual C++介绍

在Windows CE程序设计中，Embedded Visual C++是应用最为广泛的软件开发工具之一。也是最受程序开发人员欢迎，最适合进行Windows CE下进行软件开发的工具之一。或许有的读者要问，EVC和目前微软主推的Visual Studio有什么不同呢？在.NET技术出现之前，微软公司就已经为Windows CE推出了两个开发工具，一个是Embedded Visual C++，一个就是Embedded Visual Basic。从名称上可以很容易地看出，一种使用的是C++语言，一种使用的是Basic语言。

自从微软发布.NET战略以后，微软公司在推出的Visual Studio.net（Visual Studio 2005中已经没有了.NET字样）中又添加了C#语言。同时针对Smart Phone和Pocket PC等智能设备的开发作了一系列的推广工作。但是作为使用较早的EVC，已经在嵌入式程序设计领域拥有了大量的用户群体。尽管微软已经宣称不再发布Embedded Visual C++和Embedded Visual Basic的新版本，而鼓励大家使用Visual Studio，但是这一转变则需要大量的时间。本书作为一本面向基础的Windows CE程序设计教程，在考虑到大量使用人群的原则下，从EVC的使用开始讲解。那些想要了解Visual Studio程序的读者也不要着急，在本书的后面几章中会介绍采用Visual Studio.net 2003进行Pocket PC程序设计的基本方法和基本过程。如果还想对其有进一步的了解，可访问本书的网址www.deegua.com，上面有本网站推出的Visual Studio系列丛书。或者通过下面的联系方式联系作者：deegua@126.com。针对读者提出的问题，我们将及时作出答复。

14.1.1 EVC4.0特性简介

在本章中讲解的程序是采用Embedded Visual C++4.0（以下简称EVC）来开发的。因此在这里主要针对Embedded Visual C++4.0的一些特点来讲解。

Embedded Visual C++是微软公司推出针对Windows CE的可视化开发工具，同时也是Embedded Visual Tools的重要组成部分。可以利用它开发出大规模，负责的Windows CE应用程序。既然作为一款针对Windows CE的可视化开发工具，其具有很多的特点。主要包括：

- 适合嵌入式程序开发
- 具备了高效的可视化开发环境
- 更迅速的编译输出
- 更为简便的控件添加
- 允许编译后的应用程序采用延时的方式加载DLL

当打开Embedded Visual C++的界面的时候，一定会发现和VC的界面非常的相似。所不同的就是EVC中提供了特定的工具和资源来开发有针对性的嵌入式程序。

在Embedded Visual C++中主要包含下面的一些组件。

- 文本编辑器：主要用于代码的编写。
- 编译器：对程序进行编译。
- 工程工作区：主要用于工程内容的组织。
- 资源编辑器：主要用于资源的修改和设计。
- 相应的模拟器：可以安装Windows CE、Pocket PC等各种手持设备的模拟器。
- 调试器：主要用于对程序的调试，如断点测试等。
- 平台管理器：主要用于在Windows CE设备上测试编写的应用程序。
- 应用代码向导：主要作用是针对各种平台的不同应用程序创建工作框架。
- 代码浏览器：主要的作用是检查和编辑程序中的函数和类，同时可以快速地现实它们之间的关系。
- 在线文档和帮助程序。

与经常使用的Visual C++相比，EVC又具备了一些特性。表14-1列出了两者之间的对比。

表14-1 VC和EVC的对比

选 项	Visual C++	Embedded Visual C++
编译器	C/C++编译器	针对不同的芯片提供了不同的编译器。主要包括： • 针对ARM系列提供的CLARM C/C++编译器 • 针对MIPS系列提供的CLMIPS C/C++编译器 • 针对PowerPC提供的CLPPC C/C++编译器 • 针对X86系列提供的X86 C/C++编译器 • 针对SuperH系列提供的SHCL C/ C ++编译器
对MFC 类库的支持	支持微软的MFC	是标准MFC类库的一个子集
对标准模板库的支持	支持微软的标准模板库	是标准模板库的一个子集
对API函数的支持	支持Win32 API函数	是Win32 API函数的一个子集
对运行时的支持	支持标准的C运行时库	是标准C运行时库的一个子集
对CPU的支持种类	只支持X86系列	支持多种CPU，支持多种指令集
调试工具	只支持本地调试工具	还包含远程调试工具。可以在实际设备上对程序进行调试
模拟器	不包含	包含相应的模拟器
关于SDK	附带的SDK包含全部的内容	自带的SDK内容相对较少，如安装额外的程序才能得到相对全面的SDK。如只有安装了Windows CE，才可以得到相对全面的Windows CE SDK

图14-1展示了EVC的开发环境界面。

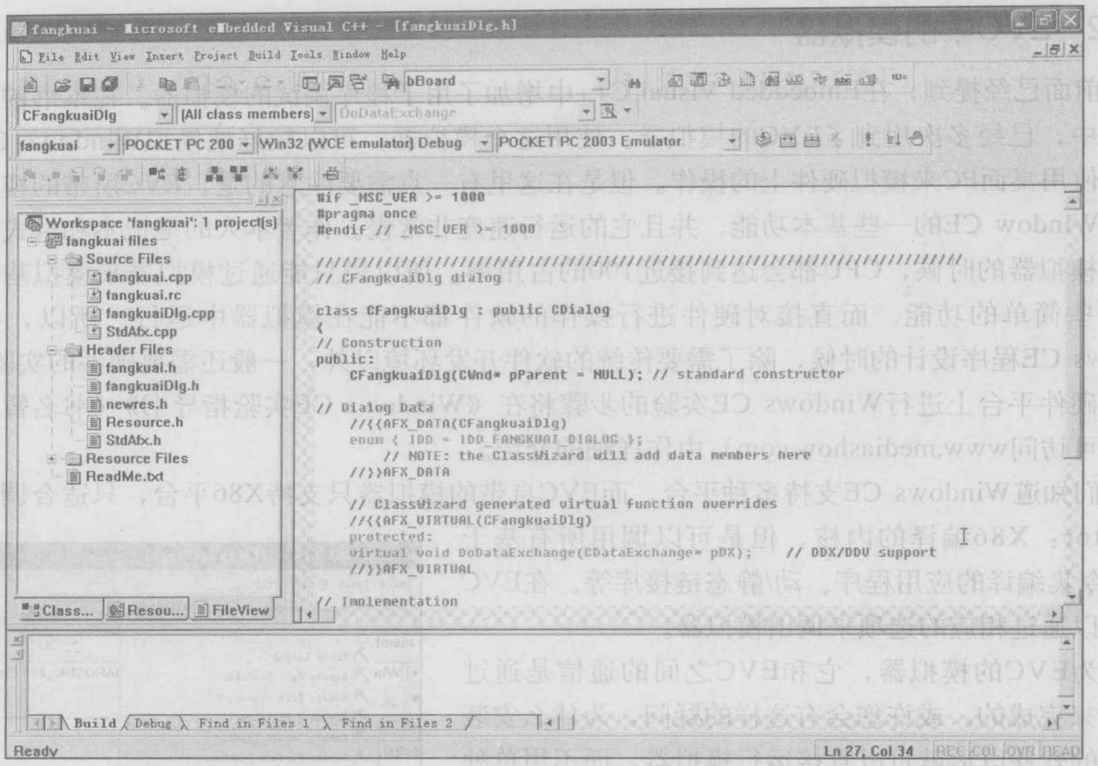


图14-1　EVC的集成开发环境界面

下面再来看看EVC中的功能菜单。图14-2展示了功能菜单。

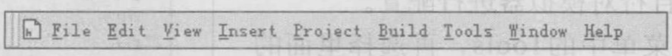


图14-2　EVC中的功能菜单

下面介绍各个菜单项的功能。

- File：可以从这里面打开、新建或者保存一个文件或者工程。
- Edit：在这里可以进行文本编辑所需要的复制、粘贴等功能。同时还提供了撤销操作和设置断点等功能。
- View：管理开发环境中的内容，如显示程序结果、资源ID、显示输出窗口、调试窗口等。
- Insert：可以在这里向当前的工程中插入类、窗体等对象。包含创建的新类，创建新窗口、创建或者倒入资源等。
- Project：在这里可以对工程进行相应的操作。主要包括设置当前项目、向当前项目中添加文件、项目的设置、插入项目到工程等。因为在一个Windows CE的工程中，可以包含很多个项目，因此设置当前项目、插入等操作就是非常实用的功能。
- Build：在这里可以对编写的应用程序进行调试。主要包括编译单个的.cpp文件，编译整个程序、重新编译整个程序，调试、设置当前平台等功能。
- Tools：在这里可以激活多种工具，如为开发Windows CE程序所需要的Remote File Viewer等。
- Window：在这里可以管理集成开发环境中的窗口。
- Help：可以通过它访问EVC的相关帮助文档。

14.1.2 EVC中的模拟器

在前面已经提到，在Embedded Visual C++中增加了用于程序调试的模拟器。在本书前面几章的讲解中，已经多次用到了EVC的模拟器。使用这个模拟器，可以较好地模拟Windows CE操作系统，使用桌面PC来模拟硬件上的操作。但是在这里有一点需要注意的是，EVC所带的模拟器只能模拟Window CE的一些基本功能，并且它的运行速度非常慢。作者本人的笔记本和台式机上运行这些模拟器的时候，CPU都会达到接近100的占用率。所以说只能通过模拟器来模拟基本的界面和一些简单的功能。而直接对硬件进行操作的软件都不能在模拟器中运行。所以，在进行Windows CE程序设计的时候，除了需要传统的软件开发环境以外，一般还需要硬件的实验平台。关于在硬件平台上进行Windows CE实验的步骤将在《Windows CE实验指导书》（书名暂定，具体信息可访问www.mediashow.com）中作详细的介绍。

我们知道Windows CE支持多种平台，而EVC自带的模拟器只支持X86平台，只适合调试基于Emulator：X86编译的内核，但是可以调用所有基于X86指令集编译的应用程序、动/静态链接库等。在EVC中，可以通过相应的选项来调出模拟器。

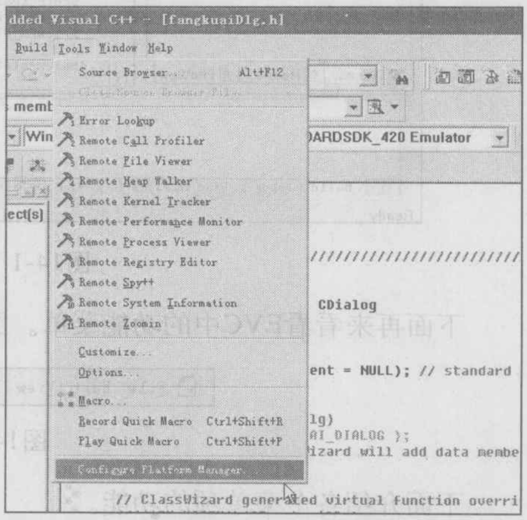


图14-3 选择模拟器配置

作为EVC的模拟器，它和EVC之间的通信是通过TCP/IP来完成的。或许您会有这样的疑问，为什么安装完相应的软件以后就可以直接运行模拟器，而不用单独配置呢？这是因为在安装相应程序的时候，EVC就会自动地设置好相应的配置，以方便程序开发人员使用。根据不同的需要，也可以自行对模拟器进行配置。

首先选择EVC功能菜单中的Tools，再选择里面的Configure Platform Manager，如图14-3所示。

此时会出现如入14-4所示的对话框。然后选择里面的STANDARDSDK_420，再双击里面的STANDARDSDK_420 Simulator，此时会出现图14-5所示的对话框。

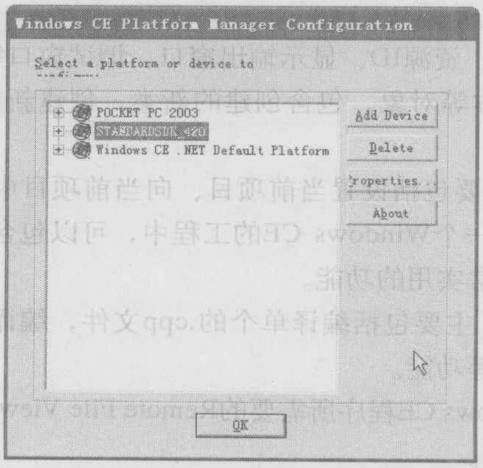


图14-4 模拟器配置1

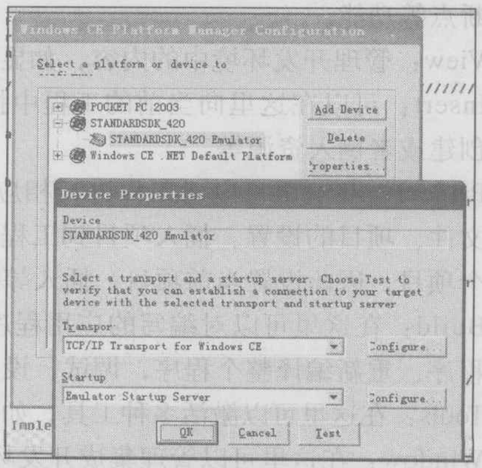


图14-5 模拟器配置2

单击图14-5中的TCP/IP Transport for Windows CE右侧的Configure按钮，会出现配置IP地址的对话框，如图14-6所示；单击Emulator Startup Server右侧的Configure按钮，会出现如图14-7所示的配置模拟器内存等信息的对话框。在这里可以更改一些设置，以建立符合要求的模拟器。

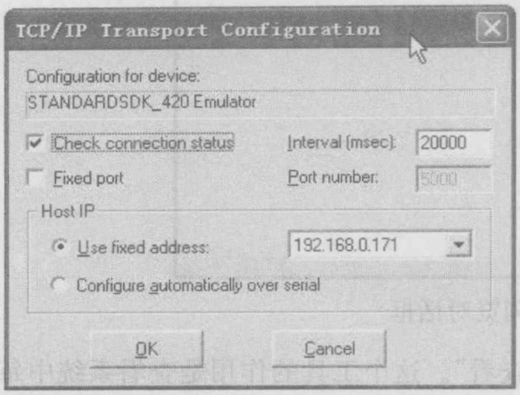


图14-6　配置对话框1

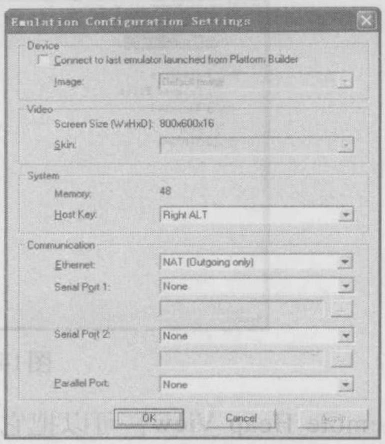


图14-7　配置对话框2

14.1.3　EVC中的远程调试工具

下面来具体地看一下EVC的远程调试工具。这些工具可以在EVC的工具栏中找到。单击工具栏上面的Tools，就会看到菜单列表中列出了EVC中的远程调试工具。图14-8展示了这些工具列表。

下面按照各个远程调试工具在列表中出现的前后顺序来作详细的介绍。

1）Remote Call Profiler：可以把它称为“远程调用评测程序”。这个程序主要包含了一些评测和分析的工具，然后利用图形图像的方式来表现程序算法的效率或者对系统资源的占用情况。但是在实际程序设计的时候，很少使用这个工具。尤其是进行非实时性内核和软件的开发设计时。

2）Remote File Viewer：可以把它称为“远程文件浏览”。这个工具是最为常用的工具之一。其作用就是从开发平台和实际应用平台之间进行文件的传输。例如，当设计的程序需要调用系统中的一个图片的时候，就需要通过这个工具把图片倒入到模拟器中。使用的方法为在菜单中选择该工具，此时会出现如图14-9所示的对话框。选择STARDARDSDK_420 Emulator，单击OK按钮

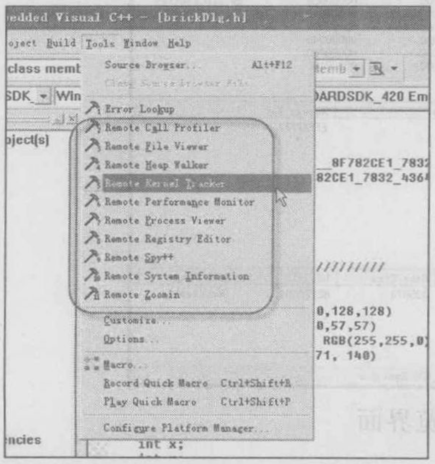


图14-8　远程调试工具

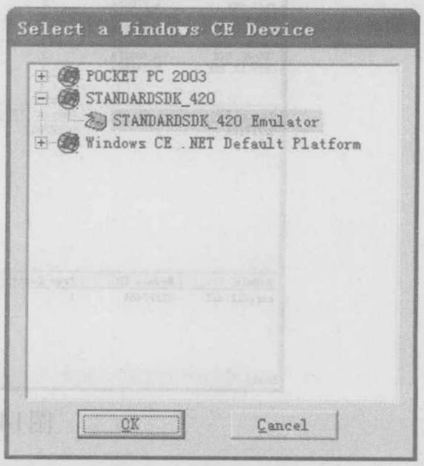


图14-9　选择模拟器

即可。在图14-10所示的对话框中就可以进行文件的导入/导出操作。

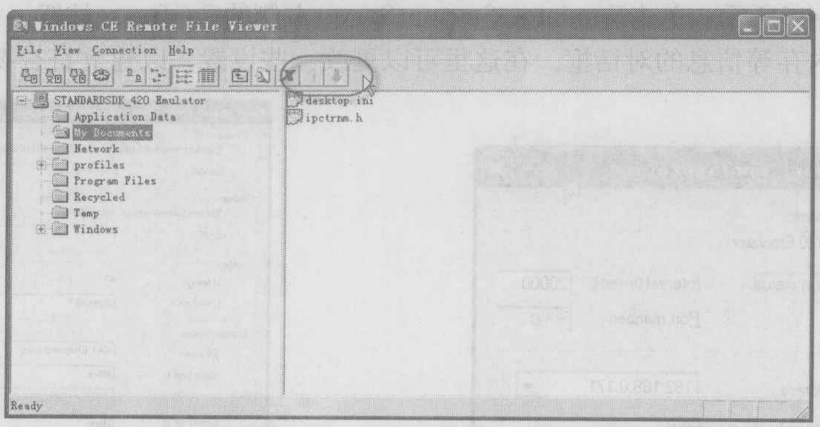


图14-10 远程文件浏览对话框

3）Remote Heap View：可以把它称为“远程堆查看”。这个工具的作用是查看系统中每个进程对堆的使用情况。例如，可以利用它查看内核中正在运程的进程的名称、ID、所使用的堆ID、进程堆中的首地址和结束地址实际内容等信息。

4）Remote Kernel Tracker：可以把它称为“远程内核跟踪”。这个工具的作用是跟踪设备中内核的执行情况。例如，可以跟踪内核启动的情况，并且把搜集到的数据在开发平台上用图标的形式表示出来。它还可以跟踪所有进程中的所有线程的使用情况，如进程的运行、阻塞、睡眠等以及线程中发生的同步事件等。

5）Remote Performance Monitor：可以把它称为“远程性能监视”。它和Windows XP下面自带的性能监视器非常类似。

6）Remote Process Viewer：可以把它称为“远程进程浏览”。它可以显示当前内核中的进程、线程和所有加载的DLL文件。图14-11为其界面。

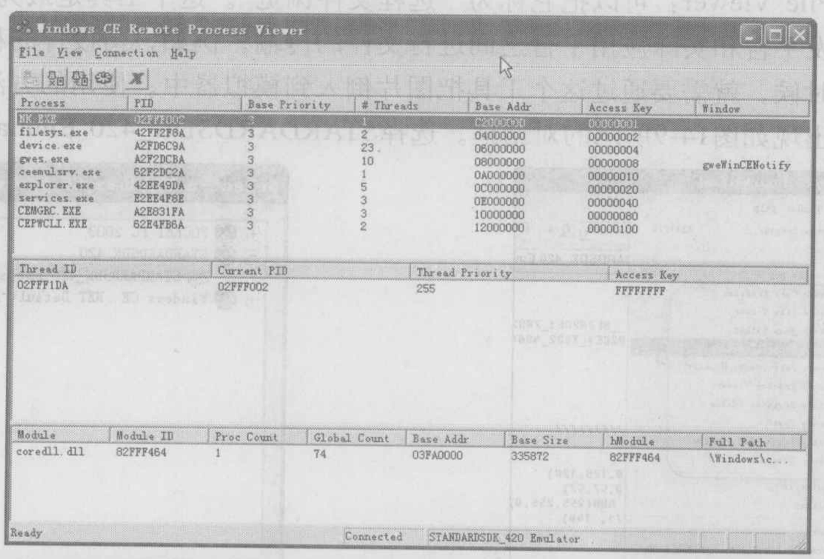


图14-11 远程进程浏览界面

从图14-11中可以看出，在表示进程的窗口中显示了进程的名称、进程的ID、进程基本的优

先级别、拥有的线程数量等信息。在显示线程的窗口中，显示了线程ID、线程优先级等信息；而在显示DLL模块的窗口中，则显示了模块名称、模块ID、基地址、大小等信息。

7）Remote Registry Editor：可以把它称为“远程注册表编辑器”。这个全程注册表编译器和其他的注册表编译器非常类似，能够同时显示和编辑开发平台中的注册表和实际设备中的一些注册表信息。图14-12展示了远程注册表编辑器的界面。

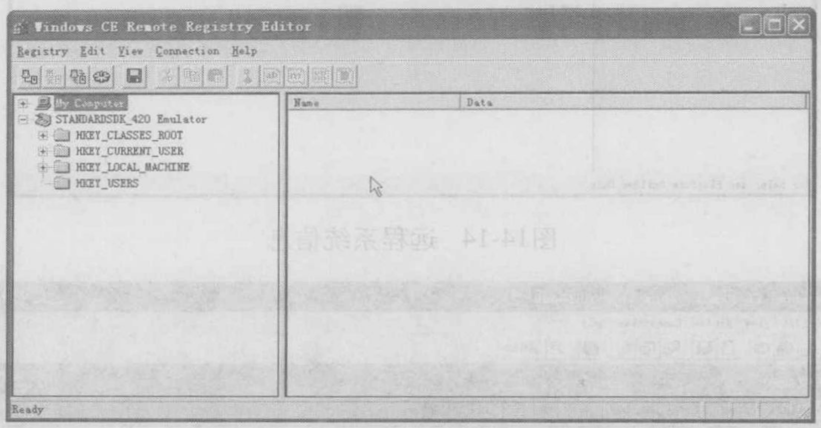


图14-12 远程注册表编辑器的界面

8）Remote Spy++：可以把它称为“远程消息监视”。使用这个工具可以监测系统发送的消息。图14-13展示了远程消息监测对话框。

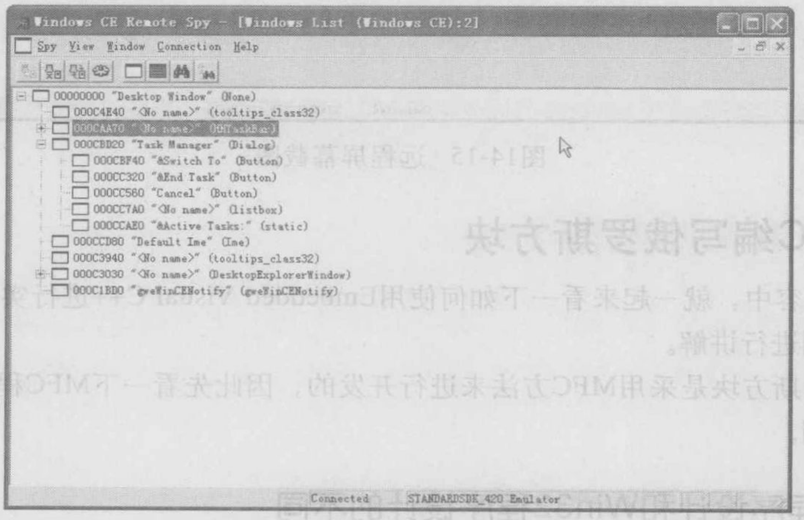


图14-13 远程消息监测对话框

9）Remote System Information：可以把它称为“远程系统信息”。使用这个工具可以查看平台的一些系统信息，如硬件信息、内存信息、端口信息等。图14-14展示了这个界面。

10）Remote Zookin：可以把它称为“远程屏幕截图”。适用这个工具可以很轻松地抓取设备屏幕，方便用户使用。在使用这个工具的时候，一般是先启动模拟器或者实际物理设备，然后启动Remote Zoomin。当这个工具运行起来以后，就会自动地截取当前设备屏幕上的画面。图14-15展示了该工具截取的Windows CE模拟器屏幕截图。

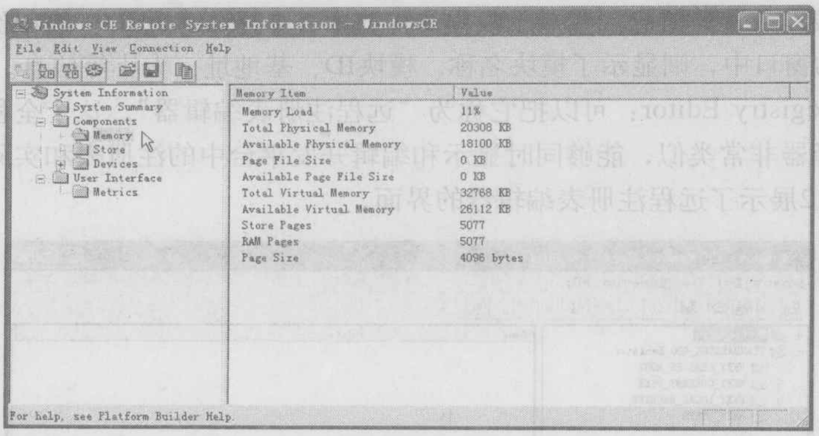


图14-14　远程系统信息

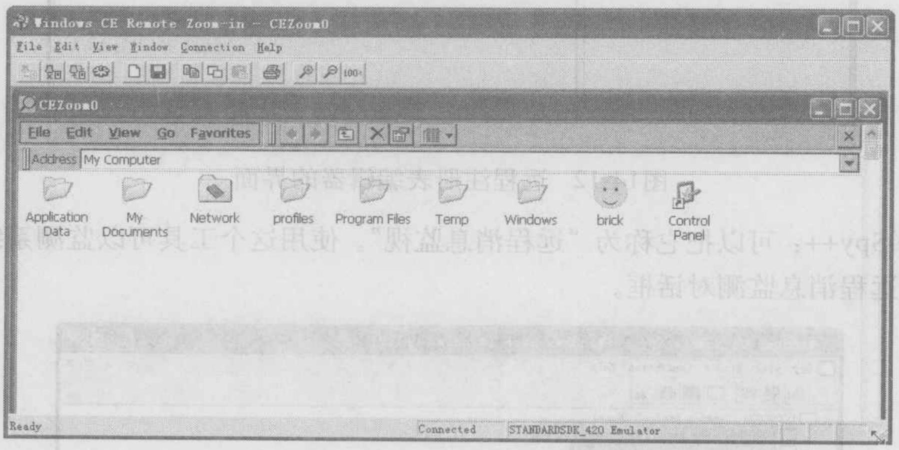


图14-15　远程屏幕截图

14.2　用EVC编写俄罗斯方块

在接下来的内容中，就一起来看一下如何使用Embedded Visual C++进行实际程序的编写。将以俄罗斯方块为例进行讲解。

由于这个俄罗斯方块是采用MFC方法来进行开发的，因此先看一下MFC程序设计和Win32程序设计有什么不同。

14.2.1　MFC程序设计和Win32程序设计的不同

在使用EVC进行程序设计的时候，通常包括Win32、MFC、ATL三种开发方法。在这里主要看一下Win32程序设计和MFC程序各自具有什么特点。

这里提到的Win32 API是Windows CE内核提供给上层应用程序、驱动程序的接口函数。在进行程序设计的时候，如果直接调用Win32 的API函数来实现这个功能，就会大大地增强软件的执行效率。同时，直接使用系统提供的API函数也可以大大地提高程序设计人员的工作效率，具有很高的灵活性。

在Windows CE中，系统的API函数从功能上主要分为服务性应用、核心操作系统服务、字体、

最终用户、硬件支持、安全、多媒体等。如果要进一步了解Windows CE中更为详细的Win32 API信息，可以查询Windows CE自带的帮助文档或者微软提供的MSDN。

与Win32 API编程方式不同，MFC方法编程的主要特点在于使用MFC库来建立应用程序的框架和对应用程序界面的编写。而程序其余的主要功能则采用Win32 API方式实现。正是由于MFC具备了这一类特点，使得很多使用Visual C++或者Embedded Visual C++开发的工具全部或者部分采用MFC来编写。因此，可以说MFC编程方式充分发挥了API设计的灵活性和MFC在界面、框架等设计方面的方便性和快捷性。

14.2.2　程序简介

在学习俄罗斯方块程序之前，先看一下这个程序在Windows CE模拟器中运行的画面。图14-16和图14-17分别为游戏启动时和运行时的画面。

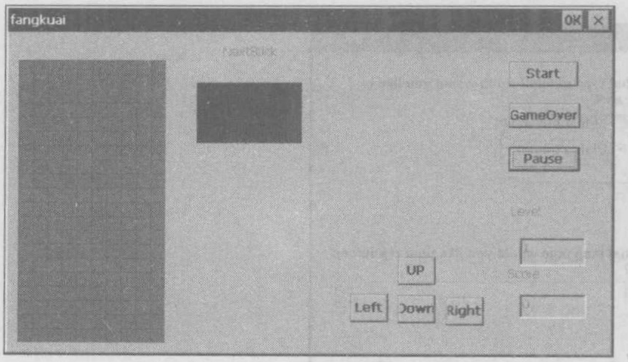


图14-16　俄罗斯方块在模拟器中启动时的画面

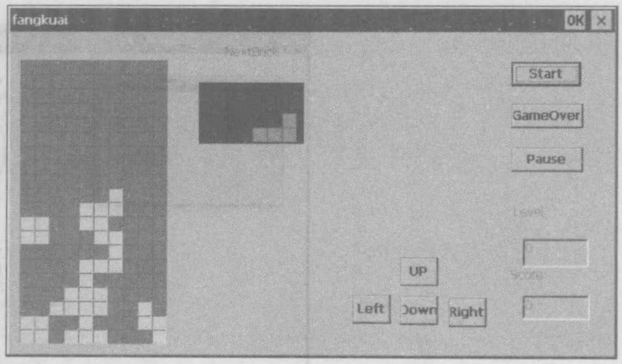


图14-17　俄罗斯方块在模拟器中运行时的画面

在程序启动的时候，游戏处于停止状态，此时可以单击右侧的Start按钮，此时游戏开始运行。由于是在模拟器中运行，可以通过鼠标单击Left、Down、Right来控制方块的左移、下降、右移的运行；通过鼠标单击UP按钮，实现方块的变形。在游戏进行的过程中，可以随时单击Pause按钮实现游戏的暂停；同时下一个将要出现的方块会在NextBrick中显示出来。

看完这个游戏以后，是否又找回了童年时的感觉了呢？下面就来看看这个程序的整个构建过程。

14.2.3　工程的建立

首先启动Embedded Visual C++4.0，选择File→New，此时会出现图14-18所示的对话框。

从图14-18中可以看到，在这个对话框的Projects选项卡中会列出使用EVC可以建立的工程。通常情况下，EVC安装完成以后是不支持建立Pocket PC程序的，只有当安装了Pocket PC相关的SDK以后，才会出现构建Pocket PC程序的选项。

- 由于在这里构建的是MFC程序，因此选择WCE MFC AppWizard（exe）选项。
- 在Project name文本框中输入想要的名字，在这里输入fangkuai。
- 在CPUs中选择Win32[WCE emulator]和Win32[WCE ARMV4]两个选项。

进行完上面一些步骤以后，单击OK按钮，进入到下一个页面。在出现的选择创建程序类型的页面中，选择Dialog based，然后单击Next按钮，如图14-19所示。

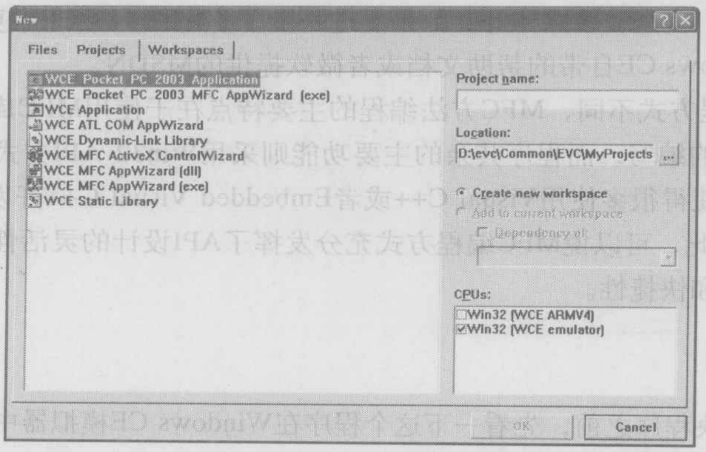


图14-18 新建工程对话框

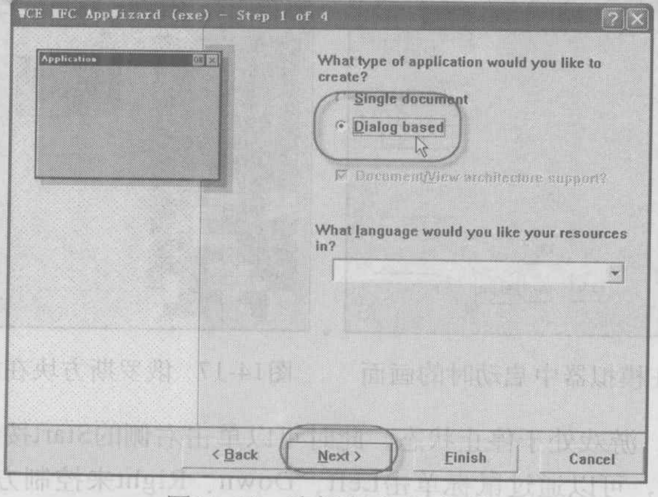


图14-19 选择程序的类型

如果对应用程序没有特殊的要求，可以在出现的图14-20所示的对话框中单击Next按钮。在出现的图14-21所示的对话框中，建议在“是否添加代码注释”的地方选择“Yes，please”。然后单击Next按钮。

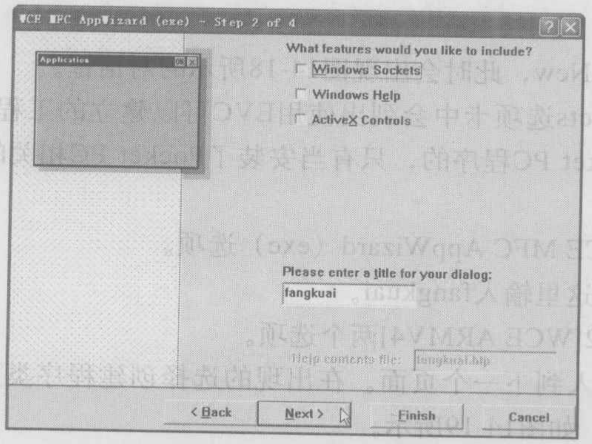


图14-20 工程构建1

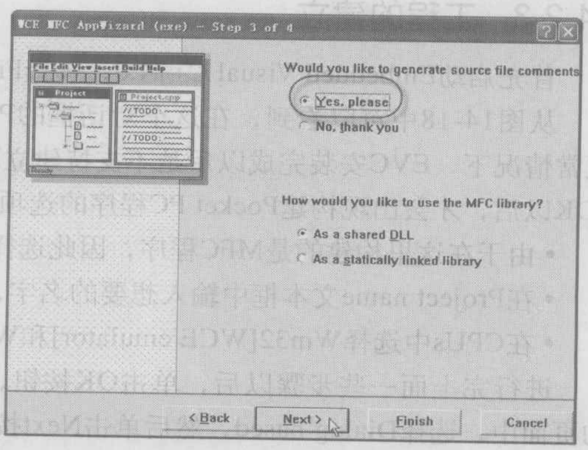


图14-21 工程构建2

在出现的对话框中，选择CFangkuaiApp，然后单击Finish按钮，如图14-22所示。

在单击Finish按钮以后，会出现显示工程信息的对话框，以提示用户工程构建完成。直接单击OK按钮即可。工程信息对话框如图14-23所示。

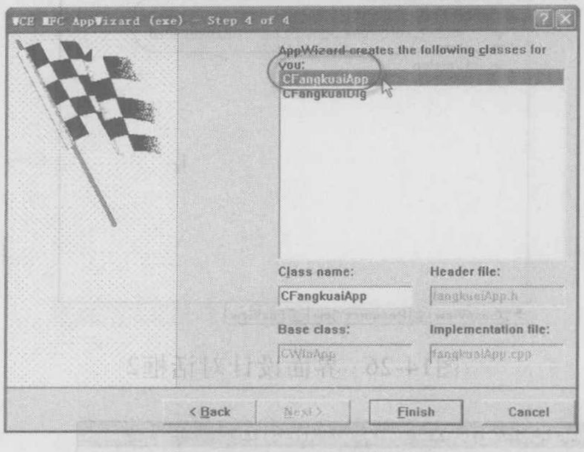


图14-22　工程构建3

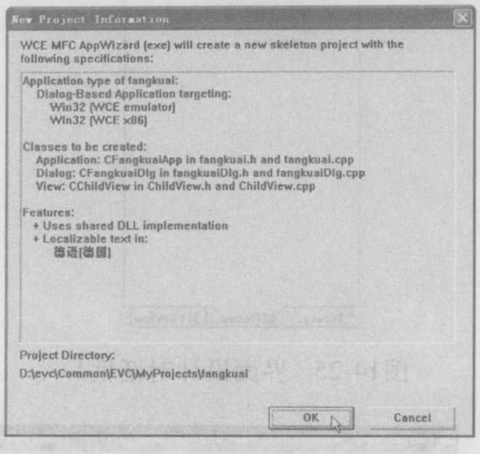


图14-23　工程信息对话框

通过上面的一系列操作就完成了一个MFC工程的构建。可以在构建好的EVC环境中看到，系统已经添加好了程序所需的基本程序框架，这也就是在前面介绍MFC程序设计特点时说的那样，使用MFC可以很方便地构建程序框架设计程序界面。关于程序界面设计上的便利，在稍后的介绍中会体会到。

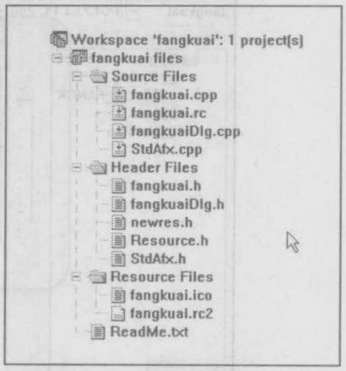


图14-24　自动生成的文件

可以编译这个工程，以调出Windows CE的模拟器，感兴趣的读者不妨尝试一下。图14-24展示了工程建立好以后EVC自动添加的文件。如果所编写的程序功能很简单，直接在相应的文件中进行代码添加和修改即可。在这里介绍的俄罗斯方块程序就是在这些已有文件的基础上，通过代码添加的方式构建起来的。

在这里需要注意的是，fangkuai.cpp、StdAfx.cpp、fangkuai.h、newres.h、Resource.h和StdAfx.h中的代码都为系统默认生成的代码；而fangkuaiDlg.cpp和fangkuaiDlg.h中的代码则为修改后的代码。关于代码的具体内容，请参考本书的附带光盘。

14.3　游戏程序界面的设计

前面已经提到，采用MFC这种程序设计方式可以较为容易地设计程序的界面。下面就看一下这个程序的界面是如何搭建起来的。

双击fangkuai.rc文件，会进入到图14-25所示的界面。此时双击Dialog，再双击Dialog下面出现的IDD_FANGKUAI_DIALOG[German[Germany]]，如图14-26所示。进行完上面的步骤以后，就会在右侧窗口出现图14-27所示的画面。

在这里的主要工作就是在图14-27标注的窗体上绘制程序相关的界面。我们会在图14-27右面看到一个工具栏，里面已经放置了一些常用的控件，包括按钮、图片控件、静态文本框等。使用的时候，只需要在控件库中选中需要添加的控件，然后在窗体的合适位置上拖动即可。如果有使

用Visual Basic的经历，一定会觉得EVC在这一点上和VB非常类似。通过控件拖放的方式就可以较为容易地设计出程序所需的画面，大大地提高了程序开发时候的效率。

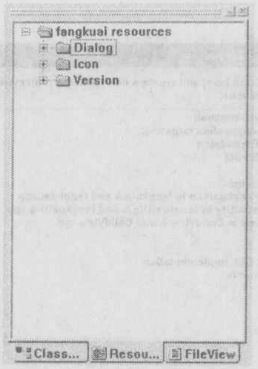


图14-25　界面设计对话框1

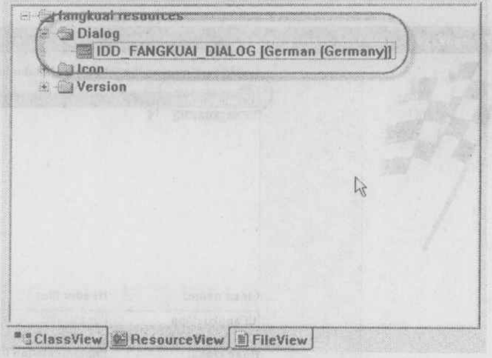


图14-26　界面设计对话框2

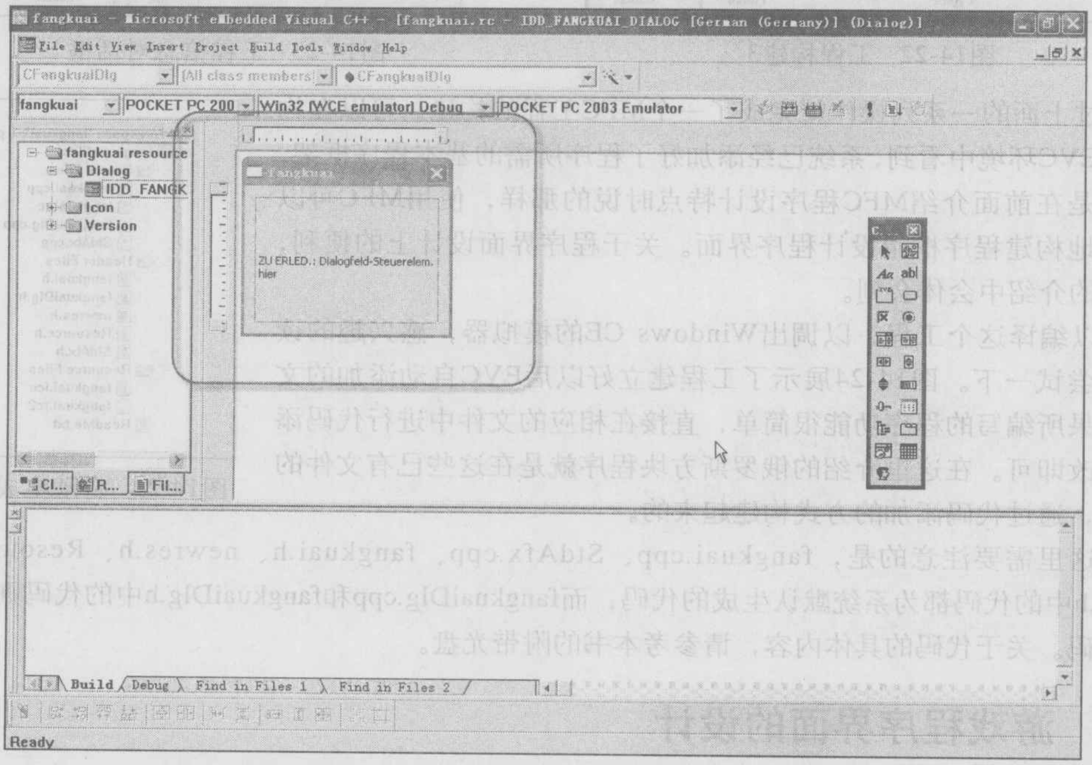


图14-27　相关程序界面

在这个程序中，一共需要7个按钮控件、5个静态文本控件。其中的7个按钮控件分别实现了程序的开始、游戏的终止、游戏的暂停、方块的左移动、方块的下移动、方块的右移动、方块的变形7个功能。5个静态文本框则实现了游戏等级显示、游戏得分显示等信息。

表14-2～表14-8列出了7个按钮相关的一些属性设置。设置按钮属性的方法为在相应的控件上单击鼠标右键，在弹出的快捷菜单中选择Properties，如图14-28所示。

在选择属性以后，会出现图14-29所示的对话框。在这里面有ID一项，这个值是该控件的标示，而Caption一项中的值则是在控件上显示的信息。

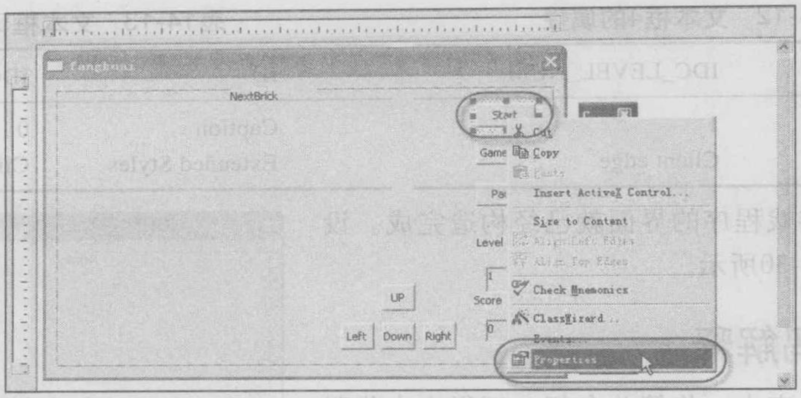


图14-28　设置控件的属性

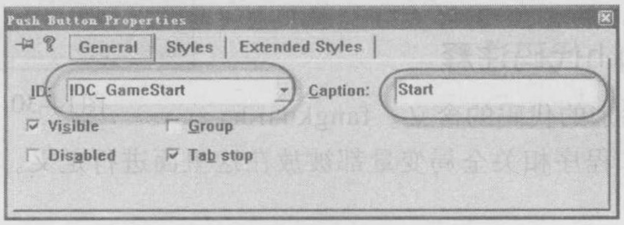


图14-29　设置控件属性对话框

表14-2　按钮1的属性

ID	IDC_GameStart
Caption	Start

表14-3　按钮2的属性

ID	IDC_GameOver
Caption	GameOver

表14-4　按钮3的属性

ID	IDC_PAUSE
Caption	Pause

表14-5　按钮4的属性

ID	IDC_LEFT
Caption	Left

表14-6　按钮5的属性

ID	IDC_DOWN
Caption	Down

表14-7　按钮6的属性

ID	IDC_RIGHT
Caption	Right

表14-8　按钮7的属性

ID	IDC_UP
Caption	Up

表14-9～表14-13列出了5个静态文本框的属性。为了更好地区分标题显示和具体的显示内容，这里对其中的两个静态文本框设置了不同的风格，请注意一下。

表14-9　文本框1的属性

ID	IDC_NEXT
Caption	NextBrick

表14-10　文本框2的属性

ID	IDC_LEVEL1
Caption	Level

表14-11　文本框3的属性

ID	IDC_SCORE
Caption	Score

表14-12 文本框4的属性

ID	IDC_LEVEL_NUM
Caption	1
Extended Styles	Client edge

表14-13 文本框5的属性

ID	IDC_STATIC_NUM
Caption	0
Extended Styles	Client edge

至此，整个游戏程序的界面就已经构造完成。设置好的界面如图14-30所示。

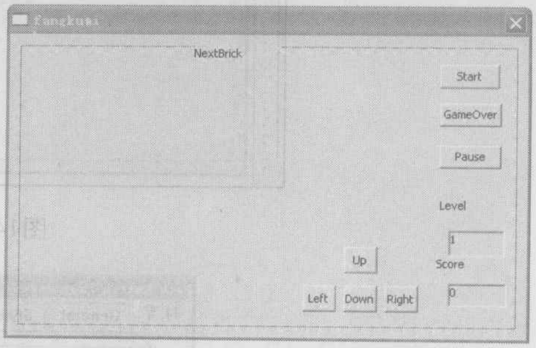


图14-30 设计好的程序界面

14.4 源代码解释

在接下来的内容中，将简单介绍一下程序中代码的含义。

14.4.1 fangkuaiDlg.h代码注释

在这里主要介绍新添加的代码的含义。fangkuai.h作为一个头文件，一些和程序相关全局变量都被放在这里面进行定义。表14-14列出了这个全局变量的含义。

表14-14 全局变量的含义

变　量	含　义
LEFTMARGIN	方块显示区域左侧距离窗体最左边的位置
TOPMARGIN	方块显示区域顶端距离窗体最上边的位置
HIGHLIGHTCOLOR	显示当前方块的区域的背景颜色
ColorOfCurrentBrick	当前显示的方块的颜色
STOP	表示程序的终止状态
RUNNING	表示程序的运行状态
PAUSE	表示程序的暂停状态
Brick	表示方块的一个结构

下面来看一下头文件中自定义的数据成员的一些含义。表14-15列出了头文件中的数据成员。

表14-15 数据成员的含义

成　员	含　义
ColorOfMatrixOfBricks	每一个方块的颜色。因为方块显示区域为20×10大小，当方块出现的时候为黄颜色，否则为背景绿颜色
EraseAline	消去一行
BrickAtBottom	方块在显示区域的底部
MatrixOfBrick	记录方块的数组。也就是方块显示的背景
GameState	游戏的状态，如运行状态、暂停状态等
CurrentBrick	当前显示的方块
LastPositionOfBrick	方块的前一个时刻的位置
TimerInterval	定时器的间隔
Level	游戏的等级
WINDTHOFBRICK	方块格子的宽度
HEIGHTOFBRICK	方块格子的高度

再来看看程序中的成员函数的意义。表14-16列出了这些成员函数所表示的含义。

表14-16　成员函数的含义

函　数	含　义	函　数	含　义
myDraw	绘制新的图形	RefreshAll	刷新整个屏幕
GenerateABrick	产生一个新的方块	Stop	处理与游戏停止相关的内容
CanEraseALine	是否可以消去一行	Pause	处理与游戏暂停相关的内容
IsGameOver	游戏是否结束	Start	处理与游戏开始相关的内容
Isbottom	方块是否落到底	Exit	处理与游戏退出相关的内容
IsOutOfRect	方块是否出了显示画面的区域	OnKey	处理键盘的输入信息
RotateBrick	旋转方块	CreateNumber	获取随机数
RefreshBricks	刷新显示方块	DrawNextBrick	绘制下一个将要出现的方块
InitBricks	初始化方块		

14.4.2　fangkuaiDlg.cpp代码注释

首先来看一下全局变量的含义。

```
    int n=0;                                  //记录分数,即消的行数
    int Number[2];                            //随机的数据用于获取方块

BOOL CFangkuaiDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);                   // Set big icon
    SetIcon(m_hIcon, FALSE);                  // Set small icon

    CenterWindow(GetDesktopWindow());   // center to the hpc screen

// TODO: Add extra initialization here
```

上面一段代码的作用是进行初始化，从最后一行添加其余初始化的提示中可以看到，自定义的那些初始化代码应该添加在此处。这也就是在前面建议把添加代码注释选择上的原因，以方便在后期对代码进行修改。

```
CRect rt;                               //声明一个矩形
GetClientRect(&rt);                     //获取客户端区域
WIDTHOFBRICKS=((rt.right-rt.left)-LEFTMARGIN)/10;
HEIGHTOFBRICKS=((rt.bottom-rt.top)-TOPMARGIN-::GetSystemMetrics(SM_CYCAPTION))/20;
WIDTHOFBRICKS=HEIGHTOFBRICKS=min(WIDTHOFBRICKS,HEIGHTOFBRICKS);
```

上面一段代码的作用是计算出方块格子的宽度和高度。同时统一格子的宽度和高度，这样就可以形成一个正方形的小格子。

```
BrickAtBottom=FALSE;                    //方块没有处在底部
```

```
    EraseALine=FALSE;               //还不能删除一行
    Level=0;                        //游戏等级为0
    TimerInterval=5000/20;          //设置时间间隔
    GameState=STOP;                 //游戏的状态初始化的时候为终止状态
    InitBricks();                   //初始化游戏方块
    RefreshAll();
    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

下面看看各个方向键对应的不同移动情况。

```
void CFangkuaiDlg::OnKey(UINT nChar)
{
    {
        switch(nChar)
        {
            case VK_NUMPAD1:                                //左方向

                if(!(IsOutOfRect(1)==2))
                {
                    RefreshBricks();
                    for(int a=0;a<4;a++)
                        CurrentBrick[a].y--;                //方块对应的列值减少
                }
                break;

            case VK_NUMPAD3:                                //右方向

                if(!(IsOutOfRect(2)==1))
                {
                    RefreshBricks();
                    for(int a=0;a<4;a++)
                        CurrentBrick[a].y++;                // 方块对应的列值增加
                }
                break;
            case VK_NUMPAD7:                                //下落

                OnTimer(1);
                break;
            case VK_NUMPAD9:                                //"上"方向键按下的时候
                RefreshBricks();
                RotateBrick();
                break;
        }

    myDraw();
```

```
    }

}

void CFangkuaiDlg::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call default
    switch(nIDEvent)
    {
    case 1:
        if(GameState==RUNNING)                    //当游戏的状态处于运行状态的时候
        {
            if(Isbottom()                         //如果未到底
            {

                RefreshBricks();                  //刷新


                for(int i=0;i<4;i++)
                    CurrentBrick[i].x++;          //方块所处的行数增加
            }
            else                                  //如果方块到底了
            {
                IsGameOver();                     //判断游戏是否结束
                CanEraseALine();                  //消去一行
                DrawNextBrick();                  //绘制下一个方块以提示
                GenerateABrick();                 //生成新的方块
                Number[0]=Number[1];
                CreateNumber();                   //获取随机的数据，以产生新的方块

            }
            myDraw();

        }
        break;

    }


    CDialog::OnTimer(nIDEvent);
}

void CFangkuaiDlg::OnPaint()
{
    CPaintDC dc(this); // device context for painting
```

```
    // TODO: Add your message handler code here

    dc.FillSolidRect(180,50,100,60,RGB(0,0,255));      //设置颜色
    EraseALine=TRUE;                                    //消去一行
    myDraw();                                           //绘制新的方块

    // Do not call CDialog::OnPaint() for painting messages
}

void CFangkuaiDlg::InitBricks()
{
    for(int i=0;i<20;i++)
        for(int j=0;j<10;j++)
                MatrixOfBricks[i][j]=0;                 //初始化整个方块
    RefreshAll();
}

void CFangkuaiDlg::RefreshBricks(void)
{
    int i;
    for(i=0;i<4;i++)
    {
        MatrixOfBricks[CurrentBrick[i].x][CurrentBrick[i].y]=0;//消去原先的方块
        LastPositionOfBrick[i].x=CurrentBrick[i].x;     //记录原先位置的行
        LastPositionOfBrick[i].y=CurrentBrick[i].y;     //记录原先位置的列
    }

}

void CFangkuaiDlg::RotateBrick()                        //方块旋转
{
    int i,j;                                            //循环变量
    int IndexOfCenter=0;                                //旋转轴的方块序号
    Brick CenterBrick;                                  //用于存放中间方块的位置
    Brick TempBrick[4];                                 //用于存放当前方块的位置
    CenterBrick.y=0;
    CenterBrick.x=0;
    if(CurrentBrick[4].x==0) return;

    for(i=0;i<4;i++)
    {
        TempBrick[i].x=CurrentBrick[i].x;
        TempBrick[i].y=CurrentBrick[i].y;
    }
    if((CurrentBrick[0].x==CurrentBrick[1].x==CurrentBrick[2].x==CurrentBrick[3].x)
        ||(CurrentBrick[0].y==CurrentBrick[1].y==CurrentBrick[2].y==CurrentBrick[3].y))
    {//如果是长棒，则以第二块为轴心，否则以第一块为轴心
```

```
        CenterBrick.x=CurrentBrick[2].x;
        CenterBrick.y=CurrentBrick[2].y;
        IndexOfCenter=2;

    }
    else
    {
        CenterBrick.x=CurrentBrick[1].x;
        CenterBrick.y=CurrentBrick[1].y;
        IndexOfCenter=1;
    }
    for(i=0;i<4;i++)
    {
        if(i!=IndexOfCenter)
        {//顺时针旋转
            CurrentBrick[i].x=TempBrick[i].y-CenterBrick.y+CenterBrick.x;
            CurrentBrick[i].y=CenterBrick.x-TempBrick[i].x+CenterBrick.y;
            if((MatrixOfBricks[CurrentBrick[i].x][CurrentBrick[i].y]==2)||
                                                    //如果碰底
                (CurrentBrick[i].y>9)||             //如果右出界
                (CurrentBrick[i].y<0)||             //如果左出界
                (CurrentBrick[i].x>19)||            //如果下出界
                (CurrentBrick[i].x<0))              //如果上出界
            {//如果已出界，则不能旋转，用备份方块还原
                for(j=0;j<4;j++)
                {
                    CurrentBrick[j].x=TempBrick[j].x;
                    CurrentBrick[j].y=TempBrick[j].y;
                }
                break;
            }
        }
    }

}

int CFangkuaiDlg::IsOutOfRect(int w)                //判断方块是否出边界
{
    if(w==2)                                        //右移
    {
        for(int a=0;a<4;a++)
        {
            if((CurrentBrick[a].y>=9)               //如果即将右移出界或碰到已有方块
                ||(MatrixOfBricks[CurrentBrick[a].x][CurrentBrick[a].y+1]==2))
            {
                return 1;                           //表示已到右边界
            }
```

```
            }
        }
        if(w==1)                                    //左移
        {
            for(int a=0;a<4;a++)
                if((CurrentBrick[a].y<=0)   //如果即将左移出界或碰到已有方块
                    ||(MatrixOfBricks[CurrentBrick[a].x][CurrentBrick[a].y-1]==2))
                    return 2;                       //表示已到左边界
        }
        return 3;                                   //未到边界
    }

    int CFangkuaiDlg::Isbottom()                    //判断方块是否到底部
    {
        BrickAtBottom=TRUE;
        for(int a=0;a<4;a++)
        {
            if(CurrentBrick[a].x==19)       //如果当前方块已经到了底端,x是纵向的坐标
            {
                for(int c=0;c<4;c++)
                    MatrixOfBricks[CurrentBrick[c].x][CurrentBrick[c].y]=2;
                    //2表示不再自由下落
                return 0;                           //到底了
            }
            if(MatrixOfBricks[CurrentBrick[a].x+1][CurrentBrick[a].y]==2)
            //如果当前方块碰到了别的方块
            {

                for(int c=0;c<4;c++)
                MatrixOfBricks[CurrentBrick[c].x][CurrentBrick[c].y]=2;
                                                    //2表示不再自由下落

            return 0;                               //到底了
            }

        }
        BrickAtBottom=FALSE;
        return 1;
    }

    void CFangkuaiDlg::CanEraseALine()              //消去行
    {
        int flagover=0;
        //int i=1;                                  //用于对等级进行判断
        TCHAR m_strValue[200];
        for(int a=19;a>=0;a--)
        {
```

```
        flagover=0;

        for(int b=0;b<10;b++)
        {
            if(MatrixOfBricks[a][b]==2)
            {
                flagover++;
            }
        }
        if(flagover==10)                                    //如果可以消去
        {
            EraseALine=TRUE;
            for(int t=a;t>0;t--)                            //方块下移
            {

                for(b=0;b<10;b++)
                {
                    MatrixOfBricks[t][b]=MatrixOfBricks[t-1][b];     //标识移位
                    MatrixOfBricks[t-1][b]=0;              //本位置置空

                }
            }
            n++;                                           //如果可以消去一行,则n值增加1
            a=a+1;                                         //重新判断这一行
        }
    }

    switch (n)
    {
    case 5:
        Level=1;
        break;
    case 10:
        Level=2;
        break;
    case 15:
        Level=3;
        break;

    case 20:
        Level=4;
        break;
    case 25:
        Level=5;
        break;
    case 30:
        Level=6;
```

```
            break;
        case 35:
            Level=7;
            break;
        case 40:
            Level=8;
            break;
        case 45:
            Level=9;
            break;
        }
```

上面这段代码的含义是对等级进行计数判断。不同的等级对应不同的n值。

```
    TimerInterval=5000/20-Level*25;              //重新设定时间间隔以适合不同等级
    SetTimer(1,TimerInterval,NULL);              //重新定义记时器
    _itow(n*100,m_strValue,10 );
     SetDlgItemText(IDC_STATIC_NUM,m_strValue);//显示分数
      _itow(CFangkuaiDlg::Level,m_strValue,10 );
     SetDlgItemText(IDC_LEVEL,m_strValue);      //显示等级数


}

void CFangkuaiDlg::IsGameOver()                  //判断游戏是否结束
{
    int a;
    for(a=0;a<9;a++)
    {
        if(MatrixOfBricks[0][a]==2)              //如果任一列到顶
        {
            GameState=STOP;
            KillTimer(1);                        //停计时器

            MessageBox(_T("Game Over!"));
            InitBricks();
            myDraw();
            return;
        }
    }
}

void CFangkuaiDlg::CreateNumber()                //初始化
{

 Number[1]=rand()%7;
}
```

```
void CFangkuaiDlg::GenerateABrick()          //产生不同类型的方块
{
    switch (Number[0])
    {
        case 1:
                CurrentBrick[0].x=0;                //一字形的方块
                CurrentBrick[0].y=4;
                CurrentBrick[1].x=0;
                CurrentBrick[1].y=5;
                CurrentBrick[2].x=0;
                CurrentBrick[2].y=3;
                CurrentBrick[3].x=0;
                CurrentBrick[3].y=2;
                CurrentBrick[4].x=1;

                break;
        case 2:
                CurrentBrick[0].x=0;
                CurrentBrick[0].y=4;
                CurrentBrick[1].x=0;
                CurrentBrick[1].y=5;
                CurrentBrick[2].x=1;
                CurrentBrick[2].y=5;
                CurrentBrick[3].x=1;
                CurrentBrick[3].y=6;
                CurrentBrick[4].x=2;

                break;
        case 3:
                CurrentBrick[0].x=1;
                CurrentBrick[0].y=4;
                CurrentBrick[1].x=1;
                CurrentBrick[1].y=5;
                CurrentBrick[2].x=0;
                CurrentBrick[2].y=5;
                CurrentBrick[3].x=0;
                CurrentBrick[3].y=6;
                CurrentBrick[4].x=3;

                break;
        case 4:
                CurrentBrick[0].x=0;
                CurrentBrick[0].y=4;
                CurrentBrick[1].x=0;
                CurrentBrick[1].y=5;
                CurrentBrick[2].x=0;
                CurrentBrick[2].y=6;
```

```
                CurrentBrick[3].x=1;
                CurrentBrick[3].y=6;
                CurrentBrick[4].x=4;

                break;
        case 5:
                CurrentBrick[0].x=1;
                CurrentBrick[0].y=4;
                CurrentBrick[1].x=1;
                CurrentBrick[1].y=5;
                CurrentBrick[2].x=1;
                CurrentBrick[2].y=6;
                CurrentBrick[3].x=0;
                CurrentBrick[3].y=6;
                CurrentBrick[4].x=5;

                break;
        case 6:
                CurrentBrick[0].x=0;
                CurrentBrick[0].y=4;
                CurrentBrick[1].x=1;
                CurrentBrick[1].y=4;
                CurrentBrick[2].x=1;
                CurrentBrick[2].y=5;
                CurrentBrick[3].x=1;
                CurrentBrick[3].y=3;
                CurrentBrick[4].x=6;

                break;
        case 0:
                CurrentBrick[0].x=0;
                CurrentBrick[0].y=4;
                CurrentBrick[1].x=0;
                CurrentBrick[1].y=5;
                CurrentBrick[2].x=1;
                CurrentBrick[2].y=4;
                CurrentBrick[3].x=1;
                CurrentBrick[3].y=5;
                CurrentBrick[4].x=0;

                break;

    }
    BrickAtBottom=FALSE;
    RefreshBricks();
    myDraw();
}
```

```
void CFangkuaiDlg::myDraw()
{
    int i;
    CDC* dc=GetDC();
    if(GameState!=RUNNING)
    {
        RefreshAll();
        return;
    }
    else
    {
        for(i=0;i<4;i++)                    //在方块数组中标明当前方块
        {
            MatrixOfBricks[CurrentBrick[i].x][CurrentBrick[i].y]=1;
        }
        if(!EraseALine)                     //如果尚未消行
        {
            if(!BrickAtBottom)
                for(i=0;i<4;i++)            // 清除移动前的方块痕迹
                {
                    dc->FillSolidRect(LEFTMARGIN+LastPositionOfBrick[i].
                    y*WIDTHOFBRICKS,TOPMARGIN+::GetSystemMetrics(SM_CYCAPTION)
                    +LastPositionOfBrick[i].x*HEIGHTOFBRICKS,WIDTHOFBRICKS,
                    HEIGHTOFBRICKS,HIGHLIGHTCOLOR);
                }
            for(i=0;i<4;i++)                //重绘正在下落的方块
            {
                dc->FillSolidRect(LEFTMARGIN+CurrentBrick[i].
                y*WIDTHOFBRICKS,TOPMARGIN+::GetSystemMetrics(SM_CYCAPTION)
                +CurrentBrick[i].x*HEIGHTOFBRICKS,WIDTHOFBRICKS,
                HEIGHTOFBRICKS,ColorOfCurrentBrick);
            }

        }
        else
        {
            RefreshAll();
            EraseALine=FALSE;
        }

    }
    ReleaseDC(dc);
}

void CFangkuaiDlg::RefreshAll()
{
    CDC* dc=GetDC();
```

```
        TRACE(L"GETDC:%x\n",dc);
        TRACE(L"CYCAPTION:%x\n",::GetSystemMetrics(SM_CYCAPTION));
        /////全部重绘//////////
        for(int x=0;x<20;x++)
        {
            for(int j=0;j<10;j++)
            {
                if((MatrixOfBricks[x][j]==1)||(MatrixOfBricks[x][j]==2))
                {
                    dc->FillSolidRect(LEFTMARGIN+j*WIDTHOFBRICKS,TOPMARGIN+::
                    GetSystemMetrics(SM_CYCAPTION)+x*HEIGHTOFBRICKS,WIDTHOFBRICKS,
                    HEIGHTOFBRICKS,ColorOfCurrentBrick);
                TRACE(L"Draw Bricks\n");
                }
                else
                {

                    dc->FillSolidRect(LEFTMARGIN+j*WIDTHOFBRICKS,TOPMARGIN+::
                    GetSystemMetrics(SM_CYCAPTION)+x*HEIGHTOFBRICKS,WIDTHOFBRICKS,
                    HEIGHTOFBRICKS,HIGHLIGHTCOLOR);
                TRACE(L"Draw Background\n");
                }
            }
        }
        ReleaseDC(dc);
        ///////////////////////
    }

    void CFangkuaiDlg::Start()                              //游戏处于开始状态
    {
        // TODO: Add your control notification handler code here
        Level=0;
        TimerInterval=5000/20;                              //时间间隔
        SetTimer(1,TimerInterval,NULL);                     //设置时间
        GameState=RUNNING;                                  //游戏的状态处于运行状态
        InitBricks();                                       //初始化方块
        GenerateABrick();                                   //产生一个方块
        SetDlgItemText(IDC_STATIC_NUM,L"0");                //显示分数
        SetDlgItemText(IDC_LEVEL,L"0");                     //显示游戏等级
        CDC*dc=GetDC();
        dc->FillSolidRect(180,50,100,60,RGB(0,0,255));      //设置颜色

    }

    void CFangkuaiDlg::Stop()                               //游戏处于停止状态
    {
        // TODO: Add your control notification handler code here
```

```
    GameState=STOP;                                  //状态设置
    KillTimer(1);                                    //停计时器
    InitBricks();

}

void CFangkuaiDlg::Exit()                            //游戏退出
{
    // TODO: Add your control notification handler code here
    EndDialog(0);

}

HBRUSH CFangkuaiDlg::OnCtlColor(CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
    HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);

    // TODO: Change any attributes of the DC here
    if(nCtlColor==CTLCOLOR_STATIC)
    {
        pDC->SelectObject(&StaticFont);
        pDC->SetTextColor(RGB(255,100,0));
    }

    // TODO: Return a different brush if the default is not desired
    return hbr;
}

BOOL CFangkuaiDlg::PreTranslateMessage(MSG* pMsg)     //系统消息的传递
{
    if (pMsg->message == WM_KEYDOWN)
    {


    // TODO: Add your message handler code here and/or call default


        switch(pMsg->wParam)
        {
           case VK_RETURN:
             return true;
           case VK_SPACE:
             return true;

            case VK_LEFT:                                //左方向
            case VK_NUMPAD4:
```

```
                nChar=VK_NUMPAD1;
                OnKey( nChar) ;
                return  true;

                case VK_RIGHT:                      //右方向
                case VK_NUMPAD6:
                nChar=VK_NUMPAD3;
                OnKey( nChar) ;
                return  true;

                case VK_DOWN:                       //下落
                case VK_NUMPAD2:
                nChar=VK_NUMPAD7;
                OnKey( nChar) ;
                return true;
                case VK_UP:                         //旋转
                case VK_NUMPAD8:
                case VK_NUMPAD5:
                nChar=VK_NUMPAD9;
                OnKey( nChar) ;
                  return  true;

            }

        }

        return CDialog::PreTranslateMessage(pMsg);

}

void CFangkuaiDlg::OnGameOver()                     //游戏结束
{
TCHAR m_strValue[200];
if(GameState==STOP)
        Exit();
      else
        Stop();
    _itow(0,m_strValue,10 );
    SetDlgItemText(IDC_STATIC_NUM,m_strValue);
    SetDlgItemText(IDC_LEVEL,m_strValue);
    CDC*dc=GetDC();
    dc->FillSolidRect(180,50,100,60,RGB(0,0,255));
    return;
}

void CFangkuaiDlg::OnGameStart()                    //游戏开始按键被按下的时候
{
```

```
 Start();
 Number[0]=rand()%7;
 Number[1]=rand()%7;
     return;

 }

 void CFangkuaiDlg::OnUp()                                    //Up键被按下的时候
 {
     if(GameState==RUNNING)

     {
         nChar=VK_NUMPAD9;
     OnKey( nChar) ;
     }
     // TODO: Add your control notification handler code here

 }

 void CFangkuaiDlg::OnDown()                                  //Down键被按下的时候
 {
     if(GameState==RUNNING)
     {nChar=VK_NUMPAD7;
 OnKey( nChar) ;
     }
     // TODO: Add your control notification handler code here

 }

 void CFangkuaiDlg::OnLeft()                                  //Left键被按下的时候
 {  if(GameState==RUNNING)
 {
     nChar=VK_NUMPAD1;
     OnKey( nChar) ;
 }


     // TODO: Add your control notification handler code here

 }

 void CFangkuaiDlg::OnRight()                                 //Right键被按下的时候
 {    if(GameState==RUNNING)
 {nChar=VK_NUMPAD3;
     OnKey( nChar) ;
 }
```

```
    // TODO: Add your control notification handler code here

}

void CFangkuaiDlg::DrawNextBrick()                          //绘制下一个方块
{
    CDC*dc=GetDC();
    int i;
if(GameState==RUNNING)
{
    switch (Number[1])                                      //绘制不同类型的方块
    {
        case 1://-----//
            CFangkuaiDlg::CurrentBrick[0].x=0;
            CFangkuaiDlg::CurrentBrick[0].y=4;
            CFangkuaiDlg::CurrentBrick[1].x=0;
            CFangkuaiDlg::CurrentBrick[1].y=5;
            CFangkuaiDlg::CurrentBrick[2].x=0;
            CFangkuaiDlg::CurrentBrick[2].y=3;
            CFangkuaiDlg::CurrentBrick[3].x=0;
            CFangkuaiDlg::CurrentBrick[3].y=2;
            CFangkuaiDlg::CurrentBrick[4].x=1;
                break;
        case 2:
            CFangkuaiDlg::CurrentBrick[0].x=0;
            CFangkuaiDlg::CurrentBrick[0].y=4;
            CFangkuaiDlg::CurrentBrick[1].x=0;
            CFangkuaiDlg::CurrentBrick[1].y=5;
            CFangkuaiDlg::CurrentBrick[2].x=1;
            CFangkuaiDlg::CurrentBrick[2].y=5;
            CFangkuaiDlg::CurrentBrick[3].x=1;
            CFangkuaiDlg::CurrentBrick[3].y=6;
            CFangkuaiDlg::CurrentBrick[4].x=2;


                break;
        case 3:
            CFangkuaiDlg::CurrentBrick[0].x=1;
            CFangkuaiDlg::CurrentBrick[0].y=4;
            CFangkuaiDlg::CurrentBrick[1].x=1;
            CFangkuaiDlg::CurrentBrick[1].y=5;
            CFangkuaiDlg::CurrentBrick[2].x=0;
            CFangkuaiDlg::CurrentBrick[2].y=5;
            CFangkuaiDlg::CurrentBrick[3].x=0;
            CFangkuaiDlg::CurrentBrick[3].y=6;
            CFangkuaiDlg::CurrentBrick[4].x=3;
```

```
            break;
        case 4:
            CFangkuaiDlg::CurrentBrick[0].x=0;
            CFangkuaiDlg::CurrentBrick[0].y=4;
            CFangkuaiDlg::CurrentBrick[1].x=0;
            CFangkuaiDlg::CurrentBrick[1].y=5;
            CFangkuaiDlg::CurrentBrick[2].x=0;
            CFangkuaiDlg::CurrentBrick[2].y=6;
            CFangkuaiDlg::CurrentBrick[3].x=1;
            CFangkuaiDlg::CurrentBrick[3].y=6;
            CFangkuaiDlg::CurrentBrick[4].x=4;

            break;
        case 5:
            CFangkuaiDlg::CurrentBrick[0].x=1;
            CFangkuaiDlg::CurrentBrick[0].y=4;
            CFangkuaiDlg::CurrentBrick[1].x=1;
            CFangkuaiDlg::CurrentBrick[1].y=5;
            CFangkuaiDlg::CurrentBrick[2].x=1;
            CFangkuaiDlg::CurrentBrick[2].y=6;
            CFangkuaiDlg::CurrentBrick[3].x=0;
            CFangkuaiDlg::CurrentBrick[3].y=6;
            CFangkuaiDlg::CurrentBrick[4].x=5;

            break;
        case 6:
            CFangkuaiDlg::CurrentBrick[0].x=0;
            CFangkuaiDlg::CurrentBrick[0].y=4;
            CFangkuaiDlg::CurrentBrick[1].x=1;
            CFangkuaiDlg::CurrentBrick[1].y=4;
            CFangkuaiDlg::CurrentBrick[2].x=1;
            CFangkuaiDlg::CurrentBrick[2].y=5;
            CFangkuaiDlg::CurrentBrick[3].x=1;
            CFangkuaiDlg::CurrentBrick[3].y=3;
            CFangkuaiDlg::CurrentBrick[4].x=6;

            break;
        case 0:
            CFangkuaiDlg::CurrentBrick[0].x=0;
            CFangkuaiDlg::CurrentBrick[0].y=4;
            CFangkuaiDlg::CurrentBrick[1].x=0;
            CFangkuaiDlg::CurrentBrick[1].y=5;
            CFangkuaiDlg::CurrentBrick[2].x=1;
            CFangkuaiDlg::CurrentBrick[2].y=4;
            CFangkuaiDlg::CurrentBrick[3].x=1;
            CFangkuaiDlg::CurrentBrick[3].y=5;
            CFangkuaiDlg::CurrentBrick[4].x=0;
```

```
                    break;

        }
            dc->FillSolidRect(180,50,100,60,RGB(0,0,255));
            for(i=0;i<4;i++)
            {
                dc->FillSolidRect(175+ CFangkuaiDlg::
                CurrentBrick[i].y*CFangkuaiDlg::WIDTHOFBRICKS,80+CFangkuaiDlg::
                CurrentBrick[i].x*CFangkuaiDlg::HEIGHTOFBRICKS,CFangkuaiDlg::
                WIDTHOFBRICKS,CFangkuaiDlg::HEIGHTOFBRICKS,RGB(255, 100, 140));
            }
        }
    }

    void CFangkuaiDlg::OnPause()                    //Pause键被按下的时候
    {
    if(GameState==RUNNING)
        {
            GameState=PAUSED;
            myDraw();

        }
        else if(GameState==PAUSED)
            GameState=RUNNING;
        // TODO: Add your control notification handler code here

    }

    int CFangkuaiDlg::OnCreate(LPCREATESTRUCT lpCreateStruct)
    {
        if (CDialog::OnCreate(lpCreateStruct) == -1)
            return -1;

        // TODO: Add your specialized creation code here
        return 0;
    }

    void CFangkuaiDlg::OnStaticNum()
    {
        // TODO: Add your control notification handler code here

    }
```

第三部分

Windows Embedded CE 6.0 高级程序设计

第15章 Windows Embedded CE 6.0中的中断处理

15.1 中断基础知识

从本书前面几章的介绍中可以知道，嵌入式软件在很多情况下是需要对外界事件迅速作出响应的，这就是嵌入式软件对实时性的要求。因此在进行嵌入式软件设计的时候，尤其是当设计的软件被应用在对实时性要求很高的场合的时候，对事件的及时响应就是必须首要解决的问题之一。

例如，当设计的软件负责监控锅炉内的压强时，若按下按钮，想知道5号锅炉内的压强，而即使此时的压强监测器正在忙于监测3号锅炉，也得必须终止操作，迅速地对5号锅炉进行监测。

在嵌入式软件中，解决响应的一个比较有效的方式就是使用中断。所谓中断，顾名思义就是让嵌入式系统的微处理器挂起当前的任务，然后迅速地转到中断事件。通常情况下，中断都是由一个硬件信号开始的。在一个嵌入式系统中，大多数的驱动串口或者网络接口的I/O芯片都需要格外地注意中断信号的发生。

本章的主要内容是介绍Windows Embedded CE 6.0中的中断处理，在了解Windows Embedded CE 6.0的中断处理机制以前，应该首先来了解一下什么是中断，中断有什么样的特点，中断的发生会导致什么样的问题出现等。

首先来看一下有关微处理器的体系结构和汇编语言的知识。

15.1.1 汇编语言与微处理器的体系结构

既然提到中断，就有必要来了解一下微处理器是如何工作的。在本章后面介绍共享数据的内容中，会用到汇编语言的简单知识，因此也需要了解一下汇编语言的简单知识。

所谓汇编语言，就是指能在微处理器上执行的机器指令的另一种表现形式，同时可以读懂这种表现形式。当微处理器在进行代码处理的时候，就需要调用汇编器把汇编语言翻译成二进制的代码。在程序执行的过程中，每一条汇编语言就对应一条机器指令。例如，当编译器开始翻译C语言的时候，就会把大多数的声明转换成微处理器能够执行的机器指令，同时大多数的C编译器也都会产生出等同于C程序的汇编语言。

在这里有一点需要特别提及的是，因为不同的微处理器采用了不同的指令集合，因此不同系列的微处理器在通常情况下使用不同的汇编语言。与之相对的是，同一种微处理器内部，每个处理器上的汇编语言几乎都是一样的。

在一些典型的微处理器内部，都有一组寄存器，也被称为通用寄存器，它的作用是存储处理器正在计算的值。例如，需要对数据进行算数运算操作之前，大多数的微处理器都会把数据存放在寄存器中。同时，对于寄存器的数量和寄存器的名称，不同的微处理器也有不同的要求。

除了提到的通用寄存器以外，大多数的微处理器还有一些专用的寄存器。例如，每一个微处理器内部都会有程序计数器，其作用是跟踪微处理器要执行的下一条指令的地址。还有一些微处

理器有一个堆栈指针，其作用是存放微处理器通用堆栈的栈顶地址。

下面就来看一下几个简单的汇编语言。

```
MOVE R1, R2
```

这条指令的含义是把寄存器R2中的值复制到寄存器R1中。在某些汇编语言中，该条指令可能以相反的方向被执行，即把寄存器R1中的值复制到寄存器R2中。因此在使用汇编语言的时候，需要首先了解类似的细节，防止出现不必要的错误。

再来看看下面的一句汇编语言：

```
ADD R1, R2
```

其作用是把寄存器R2的值加到寄存器R1上。而下面的汇编语句：

```
NOT R1
```

其含义是把寄存器R1中的值每一位都取反。

在汇编语言中，还存在着一种跳转指令，可以使用它让程序指针无条件地跳转到指令中的跳转标记处。通常情况下，汇编语言跳转标记的后面都有一个冒号。

看下面的汇编语言例子：

```
    ADD R1, R2
    JUMP  NOT_ADD
    ADD R1, R4
NOT_ADD:
    MOVE R3, R1
```

上面这段汇编语言执行的过程是：

1）把寄存器R2的值加到寄存器R1上；

2）跳转到NOT_ADD标记处；

3）把寄存器R1的值存放到寄存器R3上。

我们会发现，寄存器R4的值加到寄存器R1上这一步没有被执行。在汇编语言中，还有一种跳转指令叫做条件跳转指令，即指令在某个条件为真的情况下才会发生指令跳转。看下面的代码：

```
    SUBTRACT R1, R2
    JCOND ZERO, NOT_ADD
    ADD R3, R4
NOT_ADD:
    ......
```

上面这段汇编语言执行的过程为：

1）判断寄存器R1中的值和寄存器R2中的值是否相等；

2）如果相等，则跳转到NOT_ADD标记处；

3）如果不相等，则把寄存器R4中的值加到寄存器R3中。

如果学习过数据结构，一定还会记得在数据结构中针对堆栈有压栈和出栈的操作。在汇编语言中也提供了类似的访问堆栈的指令。在汇编语言中，用PUSH指令表示压栈，用POP指令表示出栈。

下面来看一下如果把C语言中的代码翻译成汇编语言。

假设在C语言中，有下面这样一条语句：

```
y=x+255;
```

被翻译成汇编语言以后，则成为：

```
MOVE R1, (x)
ADD R1, 255
MOVE (y), R1
```

其对应的含义为：

- 把x的值存入到寄存器R1中；
- 把255加到寄存器R1中；
- 把寄存器R1中的值存放到y中。

由于本书的重点不是介绍汇编语言，因此在这里只介绍了简单的汇编语言知识。如果想了解更为详细的关于汇编语言的内容，可参考相关的书籍。

在本章的前面曾经提到过，中断是由一个硬件信号触发开始的，那么作为嵌入式系统中的微处理器是如何判断何时开始这个中断，何时需要服务的呢？实际上，每一种芯片都有判断芯片何时需要服务的引脚。硬件工程师在进行硬件设计的时候，就已经把这些引脚放到了微处理器的上面，我们把它们称为中断请求或者IRQ。通过这种方式，微处理器通过中断请求就可以知道电路中的哪些芯片需要这些服务。通常情况下，大多数的微处理器上都有数个这样的引脚，其作用就是使得不同的芯片可以连接到微处理器上，以便请求微处理器为其提供相应的服务。

图15-1就直观地展示了这种芯片间的连接方式。

在图15-1中，和CPU端连接的引脚叫做中断请求引脚，而串口1和串口2正是通过和CPU端的中断请求引脚相连来“告诉”CPU它们需要其提供服务的。

解决了处理器知道哪个芯片有中断请求的问题以后，读者又或许会有这样的疑问，处理器又是怎样找到中断程序的呢？下面就来回答这个问题。

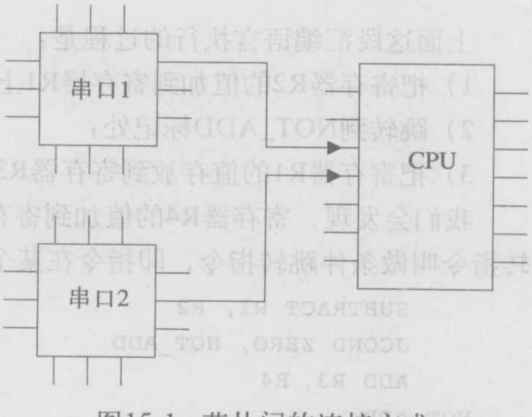


图15-1　芯片间的连接方式

实际上，不同的处理器解决这个问题的方式也会有所不同。例如，某些处理器会假定中断服务程序是存储在一个固定的位置的。如Intel的8051芯片则会默认中断程序存储在地址0x0003处，当然在这种情况下，保证中断程序必须存放在0x0003这个位置则是开发人员的事情。还有一些微处理器则采取了更为复杂的一些办法，比较典型的就是在某个位置存放一张表，这个表中的内容就是中断程序的地址，也就是经常提到的中断向量。当中断产生的时候，微处理器就会到这个中断向量表中来查找中断程序的地址。对于程序设计人员来说，在采用这种微处理器的时候，就需要很好地设计中断向量表。

提到中断向量表，又产生了一个新的问题，即微处理器如何知道中断向量表在哪里呢？同前面提到的那个问题一样，不同的处理器采取了不同的方式。有的处理器采取一个固定地址来存放这个中断向量表，而有的微处理器则通过别的一些方法把中断向量表的地址提供给用户程序。

15.1.2 中断处理程序

在这里提到的中断程序实际上就是用户写的一个子程序，当硬件捕捉到中断请求以后，就会调用该子程序。中断程序的主要功能是负责处理中断请求信号产生以后的事情。在系统运行的过程中，当微处理器检测到某个中断请求的引脚上有信号的时候，微处理器就会终止当前正在执行的指令，同时把下一条将要执行的指令地址压入到栈中，然后立即跳转到在这里提到的中断程序（interrupt routine）。

例如，当一个串口芯片从串口收到字符以后，串口芯片就会向微处理器发出一个中断请求信号，此时的中断程序就把串口芯片中的字符读到内存中。在这里需要特别提及的是，中断程序还需要处理一些其他零碎的事情，如重新启动微处理器中的中断检测硬件，其目的就是让微处理器可以检测到下一个中断。

在这里提到的中断程序有的时候还被称为中断处理程序或中断服务程序。其英文简写就是ISR（中断服务程序，Interrupt Service Routine）。

中断程序在执行的过程中，最后执行的一条汇编语句为RETURN语句。当程序执行到RETURN语句的时候，微处理器就会从堆栈中读取下一条将要执行的指令的地址（刚才已经提到，在跳转到中断程序之前，需要把下一个将要执行的指令地址压入到栈中），然后再从该地址处继续执行。

下面的一段代码很好地展示了中断程序发生时，汇编语言执行过程的变化。其中：左侧的一段代码为正常顺序执行的代码。右侧的代码为中断程序。中间的两个箭头表示从所指的位置发生中断。

```
MOVE R1, (temp)                              PUSH  R1
ADD R1, 50                                   ……
MULTIPLY  R1, 2                              POP  R1
ADD R1, 30                                   RETURN
......
```

当程序按照正常顺序执行的时候，首先把temp中的值存放到寄存器R1中，然后把50加到R1上，再把R1的值乘上2，然后再加上30……，而当中断发生在执行完R1加50以后的时候，就会首先把R1压栈，然后执行右侧中断程序的内容。当执行完以后，再把R1弹栈，重新从“MULTIPLY R1, 2”处开始执行。

15.1.3 保存和恢复上下文

在介绍什么是保存和恢复上下文之前，先来看一下当中断发生的时候，会对数据产生什么样的影响。

以15.1.2节中的汇编代码实例为例，当按照左侧的代码顺序正常执行的时候，假设temp的值为5，则顺序执行以后的效果为：

- R1=5。
- R1=5+50=55。
- R1=55*2=110。
- R1=110+30=140。

而当发生中断的时候，我们知道首先需要把寄存器R1入栈，再执行右侧的中断程序。当执行

中断程序的时候，如果修改了R1的值，则当从中断程序返回的时候，寄存器R1的值就不再是55了，自然后面执行的效果也就不再正确。

为了有效地解决上面出现的问题，有效地保证寄存器R1的值，采取了保护寄存器的方法。即在中断程序开始的时候就把重要寄存器的内容保存起来，在中断程序执行结束以后再来恢复该寄存器的值。把中断程序开始时，将所有寄存器压栈的过程称为保存上下文；与之相对应地，把中断程序执行完成时，将寄存器弹栈称为恢复上下文。

在这里有几个问题需要特别地注意：

1）通常情况下，寄存器的内容是存储在堆栈里面的。以15.1.2节中的汇编程序代码为例，中断程序在一开始的时候就是把寄存器R1的值压栈，而在返回的时候则弹栈。从这个例子中得出一个提示，那就是在写中断服务程序的时候，要把所有要使用的寄存器的值进行压栈和弹栈操作。之所以这样做，是因为我们不知道会无疑中改变哪个寄存器的值，也不知道哪个寄存器的值更为重要。

2）即使不进行保存上下文和恢复上下文的操作，也不是每次都会导致错误的发生。之所以这样讲，是因为只有当中断恰好发生在计算过程中的时候（如15.1.2节中的汇编程序代码片断），才会导致错误的发生。而大多数情况下，此类错误是不易发现的。

15.1.4 中断禁止和中断延迟

既然允许中断的发生，自然也就需要禁止中断以及实现中断的延迟。首先来看一下中断的禁止。

在实际的系统中，几乎每一个系统都会允许用户通过各种方式来禁止中断的发生。也就是说，即使I/O芯片已经向微处理器发出了中断请求，也可以通过程序来实现对请求中断的禁止响应。之所以微处理器可实现中断响应的禁止，是因为很多的微处理器都允许用户程序指定哪一个芯片上的哪一个引脚的中断请求信号被忽略。其实现方式也非常的简单，只需要用户程序向微处理器某个特定的寄存器中写入一定的值即可。通过这种方式就实现了微处理器应该响应哪些中断，应该忽略哪些中断。

介绍到这里的时候，有的读者又会有疑问：假如当嵌入式系统发生了灾难性的事故，如电源故障，外界对设备产生了强烈的破坏，又有什么方式来让系统迅速地作出响应呢？如果通过程序禁止了所有的中断，系统是不是就不会对外界信号作出反应了呢？

实际上在微处理器中，并不是所有的中断都是可以被禁止的。绝大多数的微处理器上都有一个比较特殊的输入引脚，它产生的中断是不可以被禁止的。我们把它称为不可屏蔽中断。不过在这里有一点需要特别提醒的是，不可屏蔽中断不可以和任何的代码共享数据，防止由于共享数据造成的数据出错现象的发生（关于这一点在15.1.3节中已经提到过）。不可屏蔽中断只适合用来对那些超出常规的异常事件进行处理。

可以看到，既然在微处理器中既有可以被禁止的中断，也有不可屏蔽中断，那么是否可以有选择性地禁止一部分中断呢？当两个中断同时发生的时候，微处理器会优先执行哪一个呢？

在实际的设计过程中，几乎所有的微处理器都会给每一个中断信号分配一个优先级。优先级高的中断程序就会被微处理器优先执行，自然微处理器也就会根据设置的中断优先级的高低来对中断程序进行先后的控制。同时微处理器还允许用户程序指定最低优先级中断的优先级，这样微处理器对低优先级的中断可以及时地响应。因此如果想要禁止所有的中断，当然不可屏蔽中断除

外，就可以把可接收的优先级设置得比所有中断的优先级高；相反地，如果想允许所有的中断，则可以把可接收的优先级设置得比所有中断的优先级低；当把可接收优先级设置在中间位置的时候，就可以有选择地允许或者禁止一部分中断。

既然中断程序的优先级有高有低，那么高优先级的中断程序是否可以中断低优先级的中断程序呢？答案是肯定的。也就是说，对于微处理器而言，是允许一个中断信号来打断另外一个中断程序的。对于不同的处理器，其采取的方式也不尽相同。如Intel的X86系列处理器，当进入到中断程序以后，就会关闭所有的中断，因此要想实现中断的嵌套，就必须再重新打开中断。而其他有些处理器则不需要如此复杂，中断嵌套都是自动发生的。高优先级的中断程序可以打断低优先级的中断程序，但是反过来却是不行的。也就是说，当微处理器正在执行一个高优先级的中断，即使接收到了低优先级的中断信号，微处理器也不会理睬，也会执行完高优先级中断程序以后，再去执行低优先级的。

在这里有一点需要特别提及的是，在早期的Windows CE版本中是不支持中断嵌套的。只有在Windows CE 3.0以后才加入了对中断嵌套的支持。

下面再来看看中断延迟的概念。

在这里提到的中断延迟是指系统响应一个中断所需要花费的时间。它反映了系统对每个中断响应速度的快慢。系统对于中断响应的速度主要受到下面几个因素的影响：

1）中断被禁止的最长时间；

2）在优先级较低的中断需要被执行的时候，当前优先级较高中断执行的时间；

3）微处理器停止当前的任务，保存必要信息以及执行中断程序指令所花费的时间；

4）中断程序保存上下文到完成一次响应所需要的时间。

在计算中断延迟的时候，一般会包含上面提到的4个主要因素。当然不同的算法在计算的时候会采用上面4个因素的不同组合。

针对上面提到的4个主要因素，可以通过下面的方式得到它们的数值：

1）针对第三个，因素可以通过查阅制造商提供的微处理器相关文档来了解到该因素的数值；

2）对于其他几个因素则可以通过编程测试以及查阅文档的方式获得数值。例如，可以编写一个子程序，专门负责测试执行过程所花费的时间。

15.1.5 共享数据问题的产生

前面已经提到，中断的产生会对数据造成一定的影响。首先看下面一段代码：

```
static int temp[2];
void interrupt ReadPress(void)    //这是一个中断程序
{
    temp[0]=  //从压力传感器中读出的压力值
    temp[1]=  //从压力传感器中读出的压力值
}
void main (void)
{
    int temp0,temp1;
    while(ture)
```

```
    {
        temp0=temp[0];
        temp1=temp[1];
        If (temp0! =temp1)
             //蜂鸣器鸣叫，发出提示音
    }
}
```

上面这段代码的含义就是从两个压力传感器中读取两个压力值，然后判断这两个压力值是否相等。如果不相等蜂鸣器就会鸣叫，发出提示音。

从程序代码上看，似乎没有什么错误。实际上，如果把中断产生的影响考虑进来，该段代码就会导致误报警现象的发生。例如，当中断恰好发生在图15-2所示的地方的时候，就会产生意想不到的情况。

```
static int temp[2];
    void interrupt ReadPress(void)        //这是一个中断程序
    {
        temp[0]=                          //从压力传感器中读出的压力值
        temp[1]=                          //从压力传感器中读出的压力值
    }
    void main(void)
    {
        int temp0,temp1;
        while(ture)
        {
            temp0=temp[0];
            ..............   ---->  在此处执行中断程序ReadPress
            temp1=temp[1];
            If (temp0! =temp1)
                 //蜂鸣器鸣叫，发出提示音
        }
    }
```

图15-2　中断程序发生的位置

假设原来两个压力传感器检测到的数值都是a。程序已经执行完temp0=temp[0]这条语句，此时temp0的值是a。执行完该条语句以后，恰好发生了中断程序。而此时传过来的压力值同时变成了b。也就是说temp[0]和temp[1]的值都变成了b。当中断程序执行完以后，会返回到temp1=temp[1]的位置继续执行，当然此时temp1的值就是b了。经过下面是否相等的判断以后，自然a和b不相等，因此蜂鸣器就会发出鸣叫。而实际的情况是，此时两个压力传感器检测到的数值都是b，应该处于相等的状态。

上面这个例子中遇到的问题就属于典型的数据共享问题，即由于中断的产生而导致的错误现象的发生。或许有的读者会想，刚才那个例子之所以会产生误报的情况，是因为在代码中使用了两个“中间”的变量temp0和temp1来进行判断。如果直接使用temp[0]和temp[1]的值进行判断，就不会出现刚才的问题。那么实际的情况又会如何呢？下面就具体地看一下。

看下面修改后的代码：

```
static int temp[2];
void interrupt ReadPress(void)       //这是一个中断程序
{
    temp[0]=                         //从压力传感器中读出的压力值
    temp[1]=                         //从压力传感器中读出的压力值
}
void main (void)
{
    int temp0,temp1;
    while(ture)
    {
        If (temp[0]! =temp[1])
            //蜂鸣器鸣叫，发出提示音
    }
}
```

表面上看，似乎是解决误报问题，但遗憾的是，前面提到的那个问题仍旧存在。这是因为程序代码是需要被翻译成汇编语言的，而问题就恰好出现在判断是否相等时，给temp[0]和temp[1]赋值的那个地方。判断是否相等的语句会被翻译成下面一段汇编语言：

```
MOVE R1, (temp[0])
MOVE R2, (temp[1])
SUBTRACT R1, R2
JCOND ZERO, TEMP_OK
......
```

而当中断恰好发生在图15-3所示的地方的时候，就会出现和前面一样的问题。

```
MOVE R1, (temp[0])
..................          ——→ 中断恰好发生在此处
MOVE R2, (temp[1])
SUB TRACTR1, R2
JCOND ZERO, TEMP_OK
......
```

图15-3 中断发生的位置

在这里描述的情况仍旧是初始的时候，两个压力传感器的值都是a。发生中断的时候，两个压力传感器的值都变成了b。具体执行的过程为：

1）因为初始的时候压力是a，则寄存器R1中的值就是a；

2）发生中断程序以后，R2的值变成了b；

3）中断返回以后，R1和R2的值自然也就不相等。问题和前面的一样。

在这里需要特别提醒的是，发生中断的时候，R1的值实际上是没有被更改掉的。因为中断执行完以后，会有一个恢复上下文的过程。R1的值实际上被恢复回去了。

15.1.6　共享数据问题的解决

既然中断的产生会导致共享数据问题的出现，就需要想办法来解决这一问题。解决的方法之一就是在使用共享数据的时候，禁止对中断进行响应。假设函数Open_Interrupt()表示允许使用中断，Close_Interrupt表示禁止使用中断。则15.1.5节中的代码可以修改成：

```
static int temp[2];
    void interrupt ReadPress(void)    //这是一个中断程序
    {
        temp[0]=  //从压力传感器中读出的压力值
        temp[1]=  //从压力传感器中读出的压力值
    }
    void main (void)
    {
        int temp0,temp1;
        while(ture)
        {
            Close_Interrupt () ;
            temp0=temp[0];
            temp1=temp[1];
            Open_Interrupt () ;

            If (temp0! =temp1)
            //蜂鸣器鸣叫，发出提示音
        }
    }
```

代码上作了这样的修改以后，即使会有中断请求信号的发生，微处理器也不会跳转到中断程序，也就保证了共享数据前后的一致性。

当添加了禁止中断函数以后，就会在汇编语言中出现下面的一些变化：

```
......
CI          ; 禁止中断
MOVE R1,  (temp[0])
MOVE R2,  (temp[1])
OI          ; 允许中断

SUBTRACT R1, R2
JCOND ZERO, TEMP_OK
......
```

在一些嵌入式系统的编译器提供的库函数中，通常情况下是包含允许中断和禁止中断的函数的。因此在进行嵌入式程序设计的时候，就可以通过这些函数来实现对共享数据的保护。

15.2　Windows Embedded CE 6.0中断的组成

从前面介绍的中断相关的基础知识中，可以了解到以下几个问题：

- 大多数外围设备都可以生成中断，以便从操作系统接收服务。

• 几乎任何类型的外围设备都可能使用中断作为启动操作系统的服务操作的主要方法。
• 外围设备可以产生中断或者通知中断，所以它们的设备驱动程序必须可以处理中断，以便为相应的设备服务。
• 设备向微处理器发送IRQ中断信号。其中的逻辑中断（SYSINTR）是OAL指定的IRQ的映射。

在Windows Embedded CE 6.0中，各种外设是通过中断和Windows CE的核心进行通信的。其过程为：

1）硬件设备产生硬件中断。

2）OAL（OEM Adaption Layer，原始设备制造商适配层）完成硬件中断到逻辑中断（SYSINTR）的转换。

3）系统识别逻辑中断，同时进行相应的处理。

其中OAL层的中断处理包括以下一些功能：

（1）中断初始化

由函数BOOL BSPIntrInit(void) 完成，实现的功能有为相关寄存器分配虚拟地址、相关中断的初始化、将逻辑中断与硬件中断相关联。

（2）中断使能

由函数UINT32 BSPIntrEnableIrq（UINT32 irq）完成，完成中断使能的功能，用户添加的中断需要在此添加中断使能的处理。

（3）中断禁止

由函数UINT32 BSPIntrDisableIrq（UINT32 irq）完成，完成中断禁止的功能，用户添加的中断需要在此添加中断禁止的处理。

（4）中断完成

由函数UINT32 BSPIntrDoneIrq（UINT32 irq）完成，该函数在中断处理完成后被调用，完成打开中断的功能，用户添加的中断需要在此添加相应的处理。

（5）中断激活

由函数UINT32 BSPIntrActiveIrq（UINT32 irq）完成，该函数在中断产生后被调用，完成分配系统中断号的功能，用户添加的中断需要在此添加相应的处理。

最后，当系统发现中断以后，又将分为两部分进行处理。它们分别为中断服务例程（Interrupt Service Routine，ISR）和中断服务线程（Interrupt Service Thread，IST）。

下面看一下它们各自的特点。

15.2.1　中断服务例程（ISR）

在Windows Embedded CE 6.0中，中断响应是通过中断服务例程（ISR）来实现的，它是设备所特有的。例如，不同品牌的显卡其中断服务例程就不完全相同，它一般在OAL中提供。在Windows CE中，一般使用C语言来实现对ISR的处理。

中断服务例程（ISR）一般要求短小精悍，同时对执行的效率要求也极为严格。它们只负责实现最简单的功能，即响应设备并返回一个中断标示给操作系统的核心。而对中断的实际处理则不在ISR中实现，对中断的处理被放在了中断服务线程（IST）中。在Windows CE中，ISR的主要作用是：

1）如果数据可能丢失或者被下一个中断改写，则ISR将数据从设备读取到缓冲区中。

2）ISR清除该设备上的中断条件。

3）ISR 向内核返回一个 SYSINTR。

4）内核设置供 IST 等待的中断事件。

5）调度程序调度等待的 IST 以执行后续的中断服务操作。

而中断服务例程（ISR）又分为静态ISR和可安装ISR。

- 静态ISR：静态ISR只能静态地被编译进操作系统的核心，在运行的时候是不能被改变的。同时与中断服务线程（IST）的通信只能是单向的，即由ISR到IST。静态ISR支持中断嵌套，并且还可以使用核心的堆栈。
- 可安装ISR：由内核管理程序从动态链接库（DLL）中动态地加载。同静态ISR不同，它和IST的通信则是由IST到ISR。同时多个ISR可以同一个中断请求相关联，系统会按照加载驱动程序的顺序依次对其进行调度。在可安装ISR中，共享内存的使用也是比较灵活的。

在这里还有一点需要特别注意，那就是OAL的中断服务例程（ISR）是可以调度可安装ISR的运行的。这是因为在系统中允许中断被共享。这种共享现象所对应的就是存在多个设备驱动的情况，如系统中存在多个键盘设备的驱动程序等。对于这种情况，系统将会根据各个中断服务例程（ISR）注册加载的顺序来调度这些中断服务例程。尽管这种控制方式看上去很好，但是如果对系统的实时性要求很高，就会产生一定的不利影响。

既然ISR和设备驱动程序的联系很紧密，下面就来看一下设备驱动程序加载的时候必须执行的操作：

- 注册ISR：驱动程序必须向内核注册它的 ISR，除非该驱动程序依赖常见的 OAL ISR 函数来处理它的中断。驱动程序必须向内核注册它的 ISR，以便内核在适当的物理中断发生时调用 ISR。
- 在OAL的OEMInit函数中生成从IRQ到逻辑中断SYSINTR的映射。

在这里有一点需要特别说明的是，同其他的中断一样，高优先级的中断可以抢占低优先级的中断，同时逻辑中断SYSINTR标识符通常被放在NKintr.h中声明。

15.2.2 中断服务线程（IST）

在前面提到过，中断响应由中断服务例程（ISR）来完成，而对中断的实际处理则由在这里提到的中断服务线程（IST）来实现。

中断服务线程（IST）是运行在用户空间的中断服务线程，它负责接到系统逻辑中断号之后的中断服务处理。中断服务线程（IST）在运行的时候首先要注册自己，然后再与一个系统事件关联。通常情况下，中断服务线程（IST）使用InterruptInitialize函数来注册自己，然后使用WaitForSingleObject函数来等待终端请求事件。一段时间以后，系统的调度器就会调度这个线程工作，来进行中断事务的处理。流接口驱动程序一般是在XXX_Init里面作这些事情，同时IST也被作为Device.exe的一个附属线程。

通常情况下，一个IST的具体实现过程为：

1）创建一个事件；

2）得到系统的中断号；

3）创建一个挂起的中断服务线程（IST）；

4）设置中断服务线程（IST）的优先级别；

5）调用函数InterruptInitialize通知系统注册中断；

6）恢复中断服务线程（IST），IST开始服务。

实际上，系统对一个中断进行响应，转而再进行处理的过程是需要很多模块相互配合的，如中断服务例程、OAL例程、底层支持例程等。图15-4展示了和中断处理相关的一些组件。

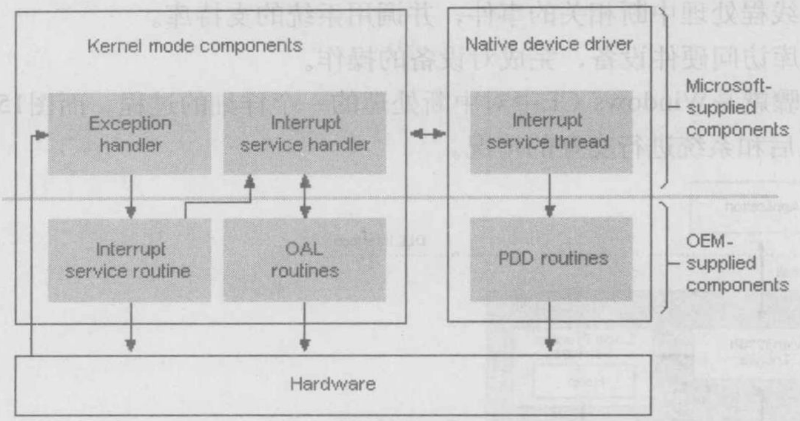


图15-4 中断服务相关的组件

15.3 Windows Embedded CE 6.0中断处理机制的特点

作为一款具有一定实时能力的嵌入式操作系统，Windows Embedded CE 6.0的ISR-IST中断处理体系结构为提高基于Windows CE的嵌入式系统的性能奠定了基础。因为一个嵌入式系统的实时处理能力不仅仅体现在系统采用了基于优先级调度的多级队列实时调度（在Windows CE中，对同级队列的调度采用了时间片轮转法）机制，还要体现在对中断处理的时间特性上。作为实施系统，自然需要快速地响应中断，然后迅速地作出处理。而Windows CE的ISR-IST中断处理机制恰好可以很好地做到这一点。

其ISR-IST中断处理机制的特点为：

- 中断服务例程（ISR）的精巧快捷的实现，有效地减少了中断屏蔽的时间。
- 系统还提供了256个优先级来进行各种系统组件以及应用程序的运行。同时，ISR-IST是运行在较高的优先级上，因此可以使得它们抢占较低优先级的组件区的时间片。
- 使用ISR-IST中断处理机制可以把关键的实时任务设置成较高的优先级。
- 对于那些较为复杂的应用，还可以实现一个较为复杂的中断服务例程（ISR）。因此可以直接在ISR中处理关键设备的工作，有效地降低了处理时间。

15.4 Windows Embedded CE 6.0中断发生和处理的过程

Windows Embedded CE 6.0中断发生和处理的过程为：

1）硬件设备产生中断信号。

2）中断信号被异常处理程序（Exception handler）捕获。

3）异常处理程序激活挂在中断服务例程（ISR）。

4）中断服务例程（ISR）响应硬件，并操作硬件设备。

5）中断服务例程（ISR）将中断映射。

6）中断服务例程（ISR）设置相应的核心中断标示。

7）中断服务调度程序与OAL例程合作，设置中断事件，操纵在特定中断事件上的等待队列。

8）OAL操作硬件，改变硬件的中断状态，将其设置为开中断。

9）中断事件激活系统调度程序，在相隔一段时间以后调度中断服务线程。

10）中断服务线程处理中断相关的事件，并调用系统的支持库。

11）系统支持库访问硬件设备，完成对设备的操作。

上面介绍的步骤就是Windows CE中对中断处理的一个详细的过程。而图15-5则从宏观上展示了硬件发生中断以后和系统进行交互的情况。

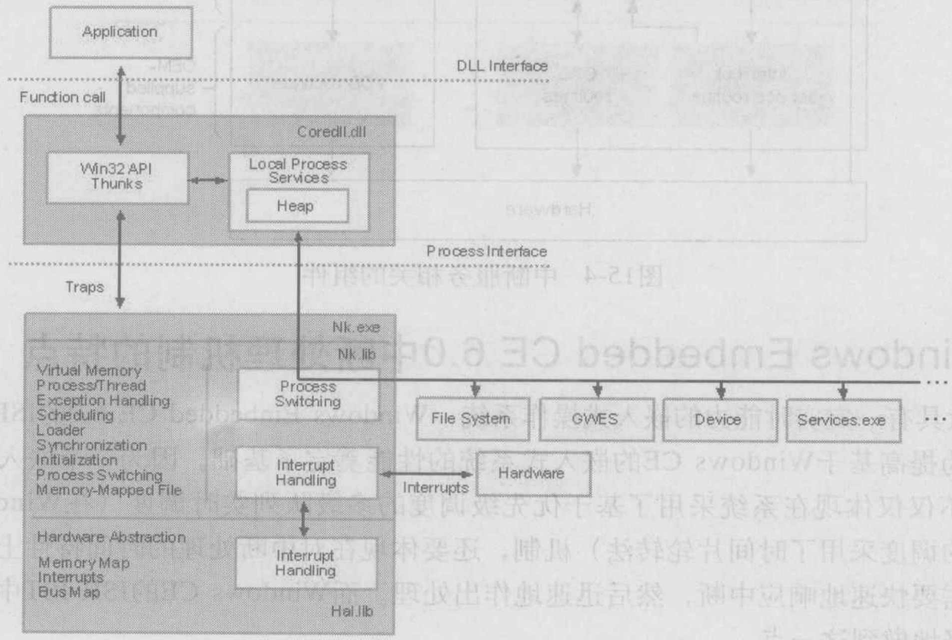


图15-5 硬件中断和系统的交互情况

在了解了Windows Embedded CE 6.0中断发生和处理的步骤以后，还有几点在这里需要特别地加以说明：

- 系统的中断响应是分阶段限制的。当核心的处理程序已经捕获了中断以后，将会禁止所有同级别的或者较低级别的中断。如果在这个过程中，有另外的级别较高的中断发生，将会出现中断嵌套的情况。
- 当中断事件被设置以后，中断服务例程（ISR）的任务就已经完成了。
- 当处理过程完成了第一阶段的任务以后，系统就没有必要再限制除同种中断之外的其他设备事件了。因此系统会开放这些中断。当最后中断处理完毕的时候，系统将会彻底地恢复到常规状态。
- 设置事件之后到中断服务线程（IST）的调度是有一个短暂的时间间隔的，也就是说并不是直接地将控制转移到中断服务线程（IST）上。
- 中断服务线程（IST）的优先级一般高于普通的线程，正是这种特性使得它们可以抢占普通线程的时间片来优先地对中断进行处理。

第16章 Windows Embedded CE 6.0中的驱动程序

16.1 Windows Embedded CE 6.0中的驱动程序模型

本章将介绍驱动程序设计的基本知识。从广义上来讲，驱动应该包含硬件和软件两个层面。驱动就是驱使物理硬件正常工作的一种方法。硬件驱动主要指的是驱动器，即提供使硬件设备可以正常工作的电流或电压的设备；而软件驱动则是指使硬件可以正常工作的一组函数接口，一方面操作了底层的硬件，另一方面提供了接口供系统调用。当然在这里提到的驱动程序主要是指软件驱动程序。

在开始介绍以前，先来看一下什么是驱动程序，驱动程序的作用是什么。

16.1.1 什么是驱动程序

所谓的设备驱动程序，实际上也是一种软件。这些软件的目的就是驱动内部和外围的硬件设备，或者为它们提供相应的接口。使用驱动程序，可以将操作系统和外围的硬件设备有机地结合在一起，使得操作系统可以很好地识别外围设备，并为应用程序提供相应的设备服务。在Windows Embedded CE 6.0中，可以把驱动程序理解为一个管理设备、服务或者协议的软件模块。设备驱动程序需要实现对特定设备的中断响应、由中断引发的数据传送和处理，以及设备的文件流接口和设备专用接口。简言之，设备驱动程序起到了沟通操作系统和硬件设备的作用。

同Windows CE 5.0不同，由于微软公司重新设计了Windows Embedded CE 6.0的驱动程序管理方式，因此在6.0版本中驱动程序既可以在核心模式下加载也可以在用户模式下加载。其中核心状态加载的方式可以使系统获得更好的性能，而用户模式加载则具有更高的安全性。

在Windows Embedded CE 6.0中，驱动程序模型可以分为本机驱动程序（Native Drivers）和流接口驱动程序（Stream Interface Drivers）。

在这里需要提及的是，不同的驱动程序模型不是通过使用它们的外围设备来区分的，而是通过它们所支持的软件接口。这些驱动程序可以是一个整体的，也可以是分层的。也就是说，驱动程序可以直接对设备进行操作，也可以把软件接口和对设备的操作放到两个层上。在Windows Embedded CE 6.0中，微软提供的许多驱动程序都采用了分层的结构。当设备被映射到系统内存上的时候，设备的驱动程序就可以直接访问它们，否则驱动程序必须通过下一层的设备驱动程序来访问外围设备。

下面首先来看一下本机驱动程序和流接口驱动程序的特点。

16.1.2 本机驱动程序

在实际的嵌入式程序设计中，经常需要把Windows CE移植到目标设备平台上，因此需要为目标平台上附带的设备（如显示器、电池、触摸屏、键盘等）提供相应的驱动程序。这类设备的驱

动程序就被称为本机驱动程序。一般只有当定制新平台的时候才需要对本机驱动程序进行开发。

本机驱动程序一般在平台启动的时候由GWES加载。OEM厂商在配置Windows CE平台的时候，一般都会把本机驱动程序与GWES链接起来，而不需要普通的用户进行添加。例如，通用LED驱动和电源驱动都是在设备出厂的时候就已经被集成在了Windows CE中，它们都是GWES的一部分，所以它们不表现在一个DLL（动态链接库）上面。其中同样LED驱动程序是由nleddrv.lib进行连接的，而电源驱动程序则是由battery.lib进行连接的。还有的如显示驱动程序和键盘驱动程序表现在一个DLL文件上，由GWES统一加以管理和加载。正如前面介绍的那样，本机设备驱动程序通常与Windows CE.Net平台紧密相连，因此本机驱动程序对于系统来说有着非常重要的作用。这些本机驱动程序的好坏，直接影响着系统的稳定性。为了提高系统的性能和安全性，微软公司通过定制接口的方式来进行对外围设备的扩展，也就是说是不需要直接和本机驱动程序打交道的。但是如果想把Windows Embedded CE 6.0移植到一个新的平台上，就需要对本机驱动程序进行相应的调整。并且在这里还有一点需要特别提及的是，本机驱动程序总是在Windows Embedded CE 6.0启动的时候进行加载的。

16.1.3　流接口驱动程序

在Windows Embedded CE 6.0中，另外一种驱动模型就是具有定制接口的流接口驱动程序。流接口驱动程序是为连接到Windows CE平台上的外围设备设计的。实际上，流接口驱动程序是一个管理外围设备的用户级的动态链接库（DLL）。它可以接受来自设备管理程序的命令和应用程序通过系统来调用的命令。在Windows CE中，流接口驱动程序几乎可以支持任何类型的连接到Windows CE系统外围设备，如打印机、调制解调器等。当然，用户还可以制定出不支持任何外围设备的流接口驱动程序。

在这里需要特别指出的是，流接口驱动程序可以由第三方硬件开发厂商设计开发，并被添加到系统中，如手持设备添加的SD卡接口外设所需要的驱动程序都要由第三方设备厂商来提供。图16-1展示了SD卡接口的摄像头，图16-2展示了SD卡接口的无线网卡。

图16-1　SD卡接口的摄像头

图16-2　SD接口的无线网卡

从前面的介绍中可以知道，在Windows Embedded CE 6.0中，当需要给它增加外围设备的时候，就必须为它提供相应的驱动程序。而这个驱动程序必须通过流接口驱动程序模型为这些新的外围设备提供驱动支持。所有的流接口驱动程序都使用同一个接口，并且调用同一个函数集（流接口函数）。这个函数集通过标准文件I/O函数来表现设备的功能。这个函数集是通过设备管理程序进行管理的。设备管理器的主要功能是在系统启动的时候，或者外围设备初始化驱动程序的时

候，通过读写注册表来进行驱动程序的加载和跟踪。当然，在不需要设备的时候就会卸载该驱动程序。例如，当向笔记本的SD卡插槽插入SD卡的时候（如笔者的IBM X40笔记本），设备管理程序就会加载SD卡的驱动程序，同时向内核注册特定的文件名。

概括地讲，本机驱动程序就是系统出厂时候自带设备的驱动程序，而流接口驱动程序则是后添加的外围设备的驱动程序。同时正是因为流驱动程序的加入，使得在驱动程序开发的时候只需要针对接口进行编程操作即可，而不用过多地去考虑驱动后面的硬件设备。

在实际应用的Windows CE设备中，还有两种重要的驱动程序，其中一个是USB驱动程序，另一个是网络驱动程序。

图16-3为USB驱动程序的整体结构。通常情况下，开发USB驱动是指Client端的驱动。图16-4展示了网络驱动程序的模型。

图16-3 USB驱动程序架构

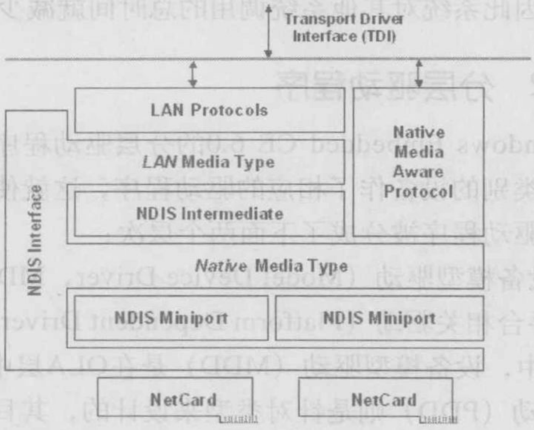


图16-4 网络驱动程序的模型

16.1.4 Windows Embedded CE 6.0中的驱动程序接口

在Windows Embedded CE 6.0中，系统是使用注册表来记录可以访问的驱动程序的信息的。并且还通过设备管理器的不同部分之间的协作来保证驱动程序可以正常、有效地被执行。既然驱动程序是针对特定设备的，就说明驱动程序一定是和某个设备相关联的。当用户访问一个设备的时候，系统会通过注册表的相关表项查询到一个设备驱动程序所提供的接口。然后Windows CE系统通过这个接口来提醒应用程序、服务和设备驱动器来关注接口的出现和消失。也就是说，在Windows Embedded CE 6.0中，系统是通过通告机制来向感兴趣的应用程序和驱动程序提示系统接口的变化的。

16.2 驱动程序的组成

在Windows Embedded CE 6.0中，设备驱动程序可以分为以下两种：

- 单体驱动程序。
- 分层驱动程序。

简单地讲，单体驱动程序就是基于单个码片，同时该码片直接把硬件设备的功能传递给操作系统；而分层驱动程序则是由两个设置好的层所组成，上层是模型设备驱动程序，下层是依赖平

台的设备驱动程序。下面就来仔细地看一下二者各有什么样的特点。

16.2.1 单体驱动程序

正如前面提到的，单体驱动程序是基于单个码片的，同时该码片直接把硬件设备的功能传递到操作系统上，因此可以把系统的一些主要功能都集成为中断处理例程，这样可以很好地提高系统的实时性。正因为这个特点，单体驱动程序一般被应用在非常关键的系统特性上。同时，一般的单体驱动程序因为消除了大部分分层驱动程序之间交互的通信代价，所以就单个设备而言，其性能得到了提高。

但是就如任何事物都有两面性一样，这种结构也会严重地影响系统的整体效率，如鼠标的驱动。如果采取这种方式，在频繁使用鼠标的时候，系统就会有非常多的时间在处理这个鼠标响应中断，因此系统对其他系统调用的总时间就减少了许多。

16.2.2 分层驱动程序

Windows Embedded CE 6.0的分层驱动程序可以适应绝大多数的设备，因为微软公司已经针对不同类别的设备作了相应的驱动程序，这就使得同类的设备可以共用大部分的代码。在分层模式中，驱动程序被分成了下面两个层次：

- 设备模型驱动（Model Device Driver，MDD）。
- 平台相关驱动（Platform Dependent Driver，PDD）。

其中，设备模型驱动（MDD）是在OLA层中实现的，和具体的设备已经进行了绑定。而平台相关驱动（PDD）则是针对类型来设计的，其目的就是保证驱动程序代码的可复用性。平台相关驱动（PDD）主要是由Windows CE系统提供支持，同时第三方的开发商也可以提供该类型的驱动程序。

设备模型驱动（MDD）通过调用平台相关驱动（PDD）中的特定例程来访问硬件的具体特性。设备模型驱动（MDD）一般提供定义设备驱动服务提供的接口，向系统提供设备驱动接口以及处理一些类似中断处理等的复杂任务。

16.2.3 单体/分层驱动程序在Windows Embedded CE 6.0中的集成

介绍完单体驱动程序和分层驱动程序各自的特点以后，再来看一下单体驱动程序和分层驱动程序是如何在Windows CE中集成的。图16-5为它们在Windows Embedded CE 6.0中的集成图示。

在这里有几个问题需要单独提及一下。在Windows Embedded CE 6.0中，模型设备驱动程序（MDD）是通用的，也就是说，MDD既是源代码也是函数库。对照图16-5会发现，模型设备驱动程序一般执行以下任务：

- 通过DDSI函数连接PDD层，并定义相应的函数。其中DDSI叫做设备驱动程序提供器接口（Device Driver Service Provider Interface）。
- 把不同的函数集提供给操作系统，通常这些函数叫做设备驱动程序接口（Device Driver Interface），简称DDI。
- 负责与GWES模块和内核之间的通信，如对中断的处理就是放在这里的。

在这里需要注意以下几个问题：

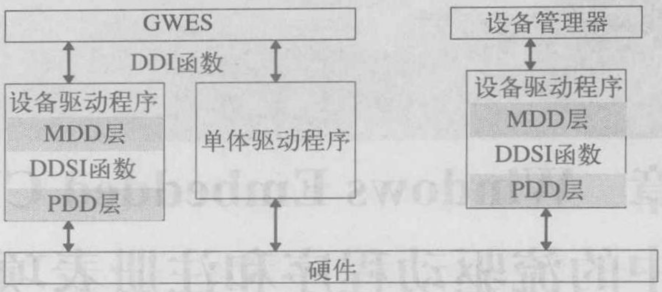


图16-5　单体驱动程序和分层驱动程序在Windows Embedded CE 6.0中的集成

- 在Windows CE 6.0中，模型设备驱动程序（MDD）一般也会负责处理音频设备或者触摸屏这样的特殊设备集。
- 设备驱动程序接口（DDI）则是一套由MDD提供的，由GWES调用的函数。
- SSDI则是由PDD层的驱动程序提供的，并由MDD调用的。

从图16-5中也可以看出，如果对系统的性能要求很高，使用单体驱动程序就比分层驱动程序要好。这是因为单体驱动程序避免了与MDD和PDD相关的函数调用过程。当硬件的性能恰好与MDD层中的函数所执行任务相匹配的时候，就可以使用单体驱动程序。既然省略了中间函数的调用过程，自然单体驱动程序的效率就比分层驱动程序要高。

在微软提供的Windows Embedded CE 6.0驱动程序开发工具Platform Builder for CE 6.0中，已经为各种内部设备提供了几个样本的PDD层。以音频驱动程序为例，可以在图16-6所示的目录中看到驱动程序使用的MDD层和PDD层。

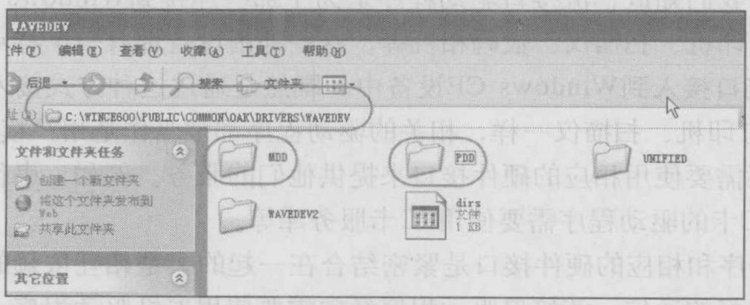


图16-6　MDD层和PDD层

在音频驱动程序MDD层中实现的是一个单独的音频设备，该设备具有播放和录制声音的功能。该驱动程序的PDD层则负责实现与相关硬件的通信，如用于开始播放声音、定制播放声音等。还包括初始化硬件和去除硬件的初始化。就函数实现而言，MDD层提供的DDI函数是标准的流接口函数，而PDD和MDD则使用一组DDSI函数通信。

第17章 Windows Embedded CE 6.0 中的流驱动程序和注册表项

17.1 流驱动程序简介

我们知道，Windows Embedded CE 6.0的驱动程序分为本机驱动程序和流驱动程序。其中本机驱动程序主要和Windows CE的自带设备相关，而流驱动程序则主要是针对系统外接设备的。那么作为针对外接设备的流接口驱动程序具有什么特点呢？本章将详细介绍。

17.1.1 什么是流接口驱动程序

在了解流驱动程序相关结构、调用过程之前，先看一下什么是流接口驱动程序。

所谓的流接口驱动程序实际上是一个动态链接库，是由一个被称为设备管理程序的特殊应用程序负责加载、管理和卸载的。同本机驱动驱动程序相比，所有的流接口驱动程序都使用同一个流接口函数集。

从前面的介绍中我们知道，流接口驱动程序是为了那些连接到Windows CE系统的外围设备准备的，如常见的打印机、扫描仪、数码相机等。这些外围设备都有一个特点，那就是都需要通过一定的外部连接接口接入到Windows CE设备中。既然是通过这种方式接入的，那么就如同桌面PC系统中的外接打印机、扫描仪一样，相关的驱动程序都被当作是用户模式的程序来运行的，同时这些驱动程序还需要使用相应的硬件接口来提供他们的服务。例如，串行设备的驱动程序需要使用串行接口，PC卡的驱动程序需要使用PC卡服务库等。

实际上，驱动程序和相应的硬件接口是紧密结合在一起的，是相互依赖的。正因为如此，为了和硬件接口进行很好的通信，流接口驱动程序经常需要使用本机驱动程序。对于那些需要连接接口（如IBM X40笔记本自带的SD卡接口、PC卡插槽等）才能连接上的设备更是如此。当然，如果外围设备被映射成了系统内存空间的一部分，那么流接口驱动程序就可以直接和外围设备进行通信，而省去了中间硬件接口的阻隔。这种直接通信的方式经常被用在Windows Embedded CE 6.0设备中的诸如喇叭、麦克风等硬件设备上。

流接口驱动程序的主要任务是把对外接设备相应的操作传递给应用程序。这个功能是通过把设备表示成文件系统的一个特殊文件来实现的。这一点和UNIX系统中把外设表示为\Dev下特殊文件的方式非常类似。尽管在Windows CE中并没有类似于\Dev的文件系统，但是也是需要把Windows CE的设备文件存在固定的路径\Windows下面的。同时还需要一个特殊的命名规则来区分这些设备文件和其他文件。

在这里有一点需要特别提及的是，尽管流接口驱动程序一般情况下是由硬件设备厂商来编写的，但是当订制基于Windows CE平台的设备的时候，也是需要自己编写一些内部设备的流接口

驱动程序的。虽然通常情况下流接口驱动程序是由设备管理程序加载和卸载的，但是仍旧可以通过一些应用程序来执行加载和卸载的任务。

17.1.2　流接口驱动程序的系统结构

本节将介绍流驱动程序的系统结构。在Windows CE中，流接口驱动程序主要接收以下两种命令：

- 从设备管理程序来的命令。
- 应用程序通过调用系统调用来的命令。

图17-1展示了流接口驱动程序和其他系统部件之间的关系。也可以在里面清楚地看到流接口驱动程序会接受到两个位置的调用命令。

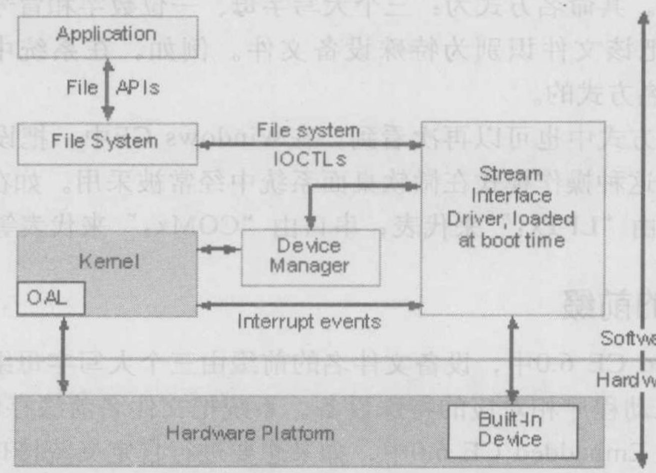


图17-1　流接口驱动程序和系统其余部件之间的关系

在这里需要对图17-1所示的各个部分作简单的说明。首先来看一下Application，即应用程序。这里所指的应用程序包括任何可以访问外围设备的应用程序。在这里，应用程序对外围设备的访问是通过文件系统和连接外围设备服务的特殊文件完成的。

图17-1中的Kernel指的是Windows CE操作系统的核心，即Windows CE的内核。内核负责重定向应用程序的文件I/O函数到适当的流接口驱动程序入口点。

在本章中已经多次提到，流接口驱动程序可以接受设备管理器的命令，这里提到的设备管理器就是图17-1所示的Device Manager。实际上，设备管理器就是一个用于加载和卸载流接口驱动程序的特殊应用程序。

而流接口驱动程序所对应的位置则是图17-1中的Stream Interface Driver。我们已经提到过，流接口驱动程序实际上就是一个动态链接库，一个负责管理外围设备的动态链接库。它通过本机驱动程序访问Windows CE系统的外接设备，或者直接访问设备。前面提到的系统喇叭就是一个直接访问的例子。

本机驱动程序Build-In Device是一个由基于Windows CE平台的设备随机提供的驱动程序。正如本章前面介绍的那样，外接设备是通过本机驱动程序管理的连接程序接口连接在设备上的，因此一些流接口驱动程序是通过本机驱动程序访问外接设备的，如系统平台中的串口、并口、PC卡插槽等。

图17-1所示的硬件平台Hardware Platform主要指的是基于Windows Embedded CE 6.0的硬件平台。基于Windows CE的平台一般是由CPU、内存（闪存）、键盘、触摸屏等组成的。流接口驱动程序一般使用平台上的I/O接口。

外围设备正是通过硬件平台和Windows CE系统建立起紧密的联系的。这些外围设备主要是打印机、扫描仪、外接键盘、外接鼠标等。

17.2 设备文件名

17.2.1 设备文件名的格式

在Windows Embedded CE 6.0中，设备文件名的格式延续了微软在MS-DOS中一贯采用的定义串口和并口的命名习惯。其命名方式为：三个大写字母、一位数字和冒号组成。通过这种命名方式，文件系统就可以把该文件识别为特殊设备文件。例如，在系统中经常见到的COM1、COM2等都是属于这种命名方式的。

实际上，从这个命名方式中也可以再次看到，在Windows CE中，把设备当成是一个特殊文件的方法是经常存在的。这种操作模式在微软桌面系统中经常被采用。如在Windows操作系统的桌面版本中，打印机设备由“LPTx：”来代表；串口由“COMx：”来代表等。

17.2.2 设备文件名的前缀

在Windows Embedded CE 6.0中，设备文件名的前缀由三个大写字母组成。这三个字母的作用是识别与特殊流接口驱动程序相对应的特殊设备。系统把文件名前缀存储在被称为Prefix的注册表键值中。在Windows Embedded CE 6.0中，如果想要进行自定义流接口驱动程序的开发，就需要自己设定这三个字母的文件名前缀。当然，它可以是任意的三个字母的组合。

在这里有一点需要特别指出的是，如果开发的驱动程序和Windows Embedded CE 6.0平台上的驱动程序属于同一个种类，就需要使用相同的前缀。如当开发了一个串行设备的驱动程序的时候，就需要使用通用的串行设备驱动程序前缀COM。讲到这里，或许有的读者会问，既然使用了Windows CE系统相同的前缀COM，那么系统如何区分子定义驱动程序和其他的系统驱动程序呢？不用担心，Windows CE系统可以根据不同的索引值把它和其他拥有相同前缀的驱动程序分开。

下面就来看一下在Windows Embedded CE 6.0中，如何通过设置文件名的索引来实现同其他设备驱动程序的区别。

17.2.3 设备文件名的索引

通常情况下，流接口驱动程序的名称是由设备文件名前缀（前面刚刚介绍过）和设备文件名索引组成的。这里面提到的索引就是指紧随文件前缀的那个数字表示符号。其作用是区分同一个流接口驱动程序管理器所管理的所有设备。一般情况下，对一个索引编号的默认值是从1~9。其中1值表示第1个设备文件名，2值则表示第2个设备文件名。如果需要10个文件名，也可以把0添加上。

在Windows Embedded CE 6.0中，设备驱动程序还可以通过另外一种方式来扩展其命名空间。采用这种命名方式可以打破0～9索引编号的限制，其表现形式为加入\$device\前缀，如COM3端

口在这种命名方式下表示为\$device\COM3。既然这种命名方式不再局限于数字0~9的限制，因此也就可以看到诸如COM15的名称，当然前面一定会有\$device\前缀。需要特别注意的是，在这种命名方式下，冒号（:）就不再需要了。

在Windows Embedded CE 6.0中还有一点需要特别注意，那就是系统中的驱动程序是通过注册表键值里面的Index值来指明起始索引数的。例如，在实际的Windows CE设备中，“COM1:”、“COM2:”和“COM3:”对应的是内部串口的硬件。当需要开发一个自定义的同样基于串口的驱动程序的时候，这个串口驱动程序同样需要以COM的前缀出现，此时可以通过指定索引值为4的方法来区别它和其余的串口设备。

17.3　流驱动的标准接口

17.3.1　常用的接口函数

在了解函数的具体功能之前，先看一下流接口驱动程序要实现的DLL接口。表17-1列出了相关的函数。

表17-1　流接口驱动程序要实现的DLL接口

函数名称	描　述
XXX_Close	驱动程序关闭时候调用
XXX_PreClose	XXX_Close函数被调用之前调用该函数
XXX_Open	打开一个设备驱动时调用
XXX_Deinit	设备管理器或者应用程序卸载驱动时调用
XXX_PreDeinit	驱动程序卸载之前被调用
XXX_Init	设备管理器初始化设备时调用
XXX_IOControl	上层软件进行IO控制调用
XXX_Read	打开设备时候进行的读操作
XXX_Write	打开设备时候进行的写操作
XXX_Seek	对设备指针进行操作时调用

上面函数中的XXX三个字母在实际的程序设计中要由设备驱动程序的设备文件名前缀所代替。例如，当编写的流接口驱动程序的设备文件名前缀为STR的时候，相应的DLL接口就变成STR _Close、STR _Open、STR _Deinit等。再如，现在有一个用于AD转换的设备，要对这个设备编写一个流接口的驱动程序，可以将此设备起名为ADC，那么在驱动程序中就必然存在一个供系统调用的函数ADC_Open。设备管理器在加载驱动程序时会调用这个函数。

17.3.2　接口函数的函数原型

XXX_Init函数的函数原型为：

```
DWORD XXX_Init( LPCTSTR pContext,   LPCVOID lpvBusContext);
```

各个参数的含义为：

- pContext：参数pContext为指向注册表的指针，说明流驱动的活动键的路径。
- lpvBusContext：参数lpvBusContext作为ActiveDeviceEx的第四个参数传递的进程映射的指针。

如果此函执行成功，则返回一个设备上下文句柄。如果执行不成功则返回零。这个句柄会被传送给XXX_Open、XXX_PowerDown、XXX_PowerUp和XXX_Deinit 函数。在这里有一点需要注意的是，这个函数不是由应用程序调用的，而是由设备管理器调用的。其作用就是初始化设备。例如，当把一个PC插入到相应插槽的时候，设备管理器就会调用这个函数进行设备的初始化。

XXX_Init函数完成的任务主要有：

- 将已安装的设备初始化为默认状态。
- 为设备驱动程序分配全局资源。
- 注册一个PC卡服务的状态回调函数。当设备状态改变时，操作系统调用这个状态回调函数。
- 映射系统内存和I/O空间到PC卡设备内存和I/O空间。
- 请求指定的回调事件的通知。
- 对于一个中断驱动设备，注册一个中断回调函数。当设备产生一个中断时，操作系统调用这个中断回调函数。

XXX_Deinit的函数原型为：

```
BOOL XXX_Deinit (DWORD hDeviceContext   );
```

参数hDeviceContext是设备上下文的句柄，由XXX_Init函数创建和返回。若这个函数执行成功返回TRUE，不成功则返回FALSE。

当用户试图停止一个设备的时候，如当PC卡从插槽中移出，设备管理器会调用这个函数。同前面的XXX_Init类似，应用程序不需要调用这个函数。相应地，应用程序可以调用DeactivateDevice函数来中止一个设备。在这个函数中，需要释放所有已分配的资源，随后中止。

XXX_Open函数的函数原型为：

```
DWORD XXX_Open(   DWORD hDeviceContext,
                  DWORD AccessCode,
                  DWORD ShareMode );
```

各个参数的含义为：

- hDeviceContext：参数hDeviceContext表示设备上下文句柄，由XXX_Init函数创建及返回。
- AccessCode：参数AccessCode表示设备的存取权限。存取权限是CreateFile传入的读写权限的组合。
- ShareMode：参数ShareMode是设备的文件共享模式。共享模式是CreateFile读写共享权限的组合。

函数返回给调用它的应用程序一个标识所打开的设备的上下文句柄。如果设备可以被多次打开，那么可以使用这个句柄来标识每次打开的设备上下文。这个标识被传送给XXX_Read、XXX_Write、XXX_Seek和XXX_IOControl函数。如果设备没有被打开，则返回零。

当此函数执行时，设备应该为每个打开上下文和对操作的准备分配足够的资源。因为这涉及将设备准备好，以便应付读写时要用到的数据结构。

XXX_Close函数的函数原型为：

```
BOOL XXX_Close( DWORD hOpenContext );
```

参数hOpenContext表示XXX_Open返回的句柄，用于标识打开的设备上下文。当函数被成功执行以后返回TRUE，否则返回FALSE。

应用程序调用CloseHandle函数来停止使用一个流接口驱动。hFile参数指定了与设备上下文相配合的句柄。作为调用CloseHand的回应，操作系统会调用XXX_Close。

当这个函数返回时，由hOpenContext指定的文件句柄就不再可用了，如果一个应用程序试图在此之后利用这个句柄执行流I/O操作，那这个操作就会失败。

XXX_Read函数的函数原型为：

```
DWORD XXX_Read(  DWORD hOpenContext,
                 LPVOID pBuffer,
                 DWORD Count );
```

各个参数的含义为：

- hOpenContext：参数hOpenContext表示被打开的设备的上下文句柄。
- pBuffer：参数pBuffer表示指向保存由设备读取的数据的缓冲区的指针。
- Count：参数Count表示从设备读入到缓冲区的字节数。

返回0表示已经到达文件结尾，返回-1表示出错，执行成功则返回字节数。在这里需要说明的是，应用程序调用ReadFile函数从设备读取数据。操作系统相应地调用这个函数。hFile参数是设备的句柄，pBuffer参数指向包含由从设备读出的数据的缓冲区，Count参数显示了应用程序想要从设备读取的字节数。

XXX_Write函数的函数原型为：

```
DWORD XXX_Write(  DWORD hOpenContext,
                  LPCVOID pBuffer,
                  DWORD Count );
```

各个参数的含义为：

- hOpenContext：参数hOpenContext表示设备上下文句柄。
- pBuffer：参数pBuffer包含要写入数据的缓冲区指针。
- Count：参数Count表示想要写入设备的数据的字节数。

当函数成功执行的时候，返回实际写入的字节数，执行失败则返回－1。应用程序使用WriteFile来向设备写入数据，操作系统则会调用这个函数来完成功能。

XXX_IOControl函数的函数原型为：

```
BOOL XXX_IOControl(  DWORD hOpenContext,
                     DWORD dwCode,
                     PBYTE pBufIn,
                     DWORD dwLenIn,
                     PBYTE pBufOut,
                     DWORD dwLenOut,
                     PDWORD pdwActualOut );
```

各个参数的含义为：

- hOpenContext：参数hOpenContext表示设备上下文。
- dwCode：参数dwCode表示I/O的控制操作码。这些编码是由设备决定的，一般都会包含在头文件里。
- pBufIn：参数pBufIn为指向输入缓冲区的指针。

- dwLenIn：参数dwLenIn为要写入的字节数。
- pBufOut：参数pBufOut为输出缓冲区的指针。
- dwLenOut：参数dwLenOut为输出的字节数。
- pdwActualOut：参数pdwActualOut为实际输出的字节数。

该函数成功执行以后返回TRUE，否则为FALSE。需要指出的是，应用程序调用Device-IOControl函数来指定一个操作。操作系统对应地调用XXX_IOControl函数来完成这个任务。dwCode参数包含要执行的输入或输出操作。这些编码通常是由每个设备驱动程序来指定的，并且通过头文件提供给编程人员。

17.3.3 与应用程序之间的调用关系

在前面的介绍中一定会注意到，很多函数都是应用程序无法直接调用的。那么这些流接口驱动程序函数和应用程序之间又是怎样一种调用关系呢？下面就来看一下。

图17-2展示了流接口驱动程序函数和应用程序之间的调用关系。

在前面已经多次提到过，在Windows CE中，设备被当作是一种特殊的文件，应用程序对设备文件的操作都被转化为对驱动程序的请求。在Windows Embedded CE 6.0中，对设备文件的操作主要涉及下面几个函数：

- 创建文件（打开文件）。
- 读文件。
- 写文件。
- 文件指针定位。
- 关闭文件。

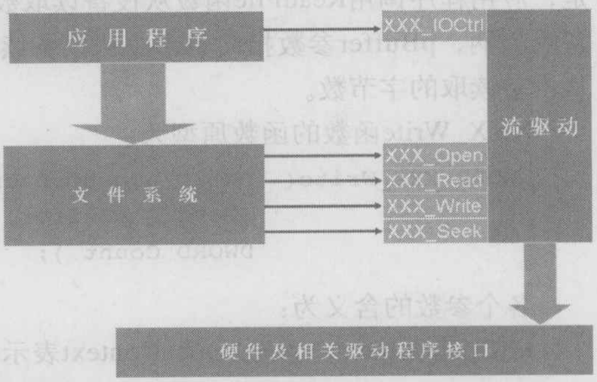


图17-2 应用程序和流接口驱动程序相关函数之间的调用关系

上述设备文件操作与设备IO的对应关系为：

- 创建文件（打开文件）==〉打开设备。
- 读文件 ==〉读设备。
- 写文件 ==〉写设备。
- 文件指针定位 ==〉设备指针定位。
- 关闭文件 ==〉关闭设备。

17.3.4 文件操作和设备操作函数参数间的对应关系

接下来，通过两个函数的对比，来看一下打开文件函数和打开设备函数之间参数的对应关系：

```
HANDLE CreateFile(  LPCTSTR lpFileName,
                    DWORD dwDesiredAccess,
                    DWORD dwShareMode,
                    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                    DWORD dwCreationDispostion,
                    DWORD dwFlagsAndAttributes,
                    HANDLE hTemplateFile );
```

```
DWORD XXX_Open(     DWORD hDeviceContext,
                    DWORD dwAccess,
                    DWORD dwShareMode);
```

上面的两个函数分别为打开文件的CreateFile函数和打开设备的XXX_Open函数，其参数的对应关系为：

- dwDesiredAccess与dwAccess相对应。
- dwShareMode与dwShareMode相对应。
- xxx_Open()返回一个无符号长整型，这个值被作为设备句柄返回给应用程序。

接下来看一下读文件的ReadFile函数和读设备的XXX_Read函数：

```
BOOL ReadFile( HANDLE hFile,
               LPVOID lpBuffer,
               DWORD nNumberOfBytesToRead,
               LPDWORD lpNumberOfBytesRead,
               LPOVERLAPPED lpOverlapped );

DWORD XXX_Read(  DWORD hOpenContext,
                 LPVOID pBuffer,
                 DWORD Count );
```

其参数的对应关系为：

- hFile与hOpenContext相对应。
- lpBuffer与pBuffer相对应。
- nNmberOfBytesToRead 与Count相对应。
- XXX_Read()返回一个无符号长整型，这个值被作为实际读取的字节数返回给应用程序。
- 如果XXX_Read()返回－1，那么表示此次操作失败，对应的ReadFile会返回FALSE。

写文件的WriteFile函数和写设备的XXX_Write函数对比：

```
BOOL WriteFile( HANDLE hFile,
                LPCVOID lpBuffer,
                DWORD nNumberOfBytesToWrite,
                LPDWORD lpNumberOfBytesWritten,
                LPOVERLAPPED lpOverlapped  );

DWORD XXX_Write( DWORD hOpenContext,
                 LPCVOID pBuffer,
                 DWORD Count );
```

其参数的对应关系为：

- hFile与hOpenContext相对应。
- lpBuffer与pBuffer相对应。
- nNmberOfBytesToWrite 与Count相对应。
- XXX_Write()返回一个无符号长整型，这个值被作为实际写入的字节数返回给应用程序。
- 如果XXX_Write()返回－1，那么表示此次操作失败，对应的WriteFile会返回FALSE。

文件指针定位函数SetFilePointer和设备指针定位函数XXX_Seek的对比：

```
DWORD SetFilePointer(     HANDLE hFile,
                          LONG lDistanceToMove,
                          PLONG lpDistanceToMoveHigh,
                          DWORD dwMoveMethod);

DWORD XXX_Seek( DWORD hOpenContext,
                long Amount,
                WORD Type );
```

其参数的对应关系为：

- hFile与hOpenContext相对应。
- lDistanceToMove与Amount相对应。
- dwMoveMethod与Type相对应。
- XXX_Seek()返回一个无符号长整型，这个值被作为SetFilePointer的返回值返回给应用程序。
- SetFilePointer的第三个参数lpDistanceToMoveHigh并不被支持，置为NULL。

作为一种更为灵活的设备处理请求，流接口提供了IOControl函数。使用此函数可以不限于读或者写，可以通过此函数处理读写请求，也可以处理一些其他任意的请求，例如设置设备参数等。

DeviceIoControl的函数原型为：

```
BOOL DeviceIoControl( HANDLE hDevice,
                      DWORD dwIoControlCode,
                      LPVOID lpInBuffer,
                      DWORD nInBufferSize,
                      LPVOID lpOutBuffer,
                      DWORD nOutBufferSize,
                      LPDWORD lpBytesReturned,
                      LPOVERLAPPED lpOverlapped);
```

XXX_IOControl的函数原型为：

```
BOOL XXX_IOControl( DWORD hOpenContext,
                    DWORD dwCode,
                    PBYTE pBufIn,
                    DWORD dwLenIn,
                    PBYTE pBufOut,
                    DWORD dwLenOut,
                    PDWORD pdwActualOut    );
```

从上面两个函数中可以看到，两者的参数也是一一对应的。

17.3.5 单访问和多访问

在Windows Embedded CE 6.0中，流接口驱动程序还可以实现单访问和多访问。简单地讲，单访问是指只有第一个打开设备的应用程序可以访问该设备，相对应地，多访问则是指对于每个打开该设备的应用程序，都可以随时访问设备而互不干扰。之所以会出现单访问和多访问的情况，是因为在Windows Embedded CE 6.0中，外围设备是以特殊文件的形式供应用程序使用的。

要想在Windows CE设备中实现单访问和多访问，可以通过使用hOpenContext参数来实现。具体的实现方法为：

- 实现单访问：只有第一次对XXX_Open的调用返回一个合法的hOpenContext的值。直到系统调用XXX_Close得到新值以前，hOpenContext的值一直是合法的。所有后来对XXX_Open的调用都会返回一个空值NULL。
- 实现多访问：和实现单访问时不同，每一次对XXX_Open的调用都返回不同的hOpenContext的值。设备驱动程序会跟踪从XXX_Open返回的值中哪一个正在被使用。当然，在这种情况下系统对诸如XXX_Close、XXX_Read、XXX_Write的调用也都需要明确哪个hOpenContext正在访问这些函数，从而使驱动程序可以很好地识别要处理的内部数据结构。

17.4 样本流接口驱动程序

在Windows Embedded CE 6.0中，平台的建立程序已经内置了一些类型的流接口驱动程序样本。因此当需要编写一种与样本驱动程序类似的流接口驱动程序的时候，就可以在这些样本驱动程序源代码的基础上，进行相应代码的改写。这样就会大大地节省开发所需的时间。例如，在使用Platform Builder for CE 6.0定制的操作系统镜像的过程中，可以选择添加USB驱动程序、SD卡驱动程序、串口驱动程序、PC卡驱动程序等。

17.5 Windows Embedded CE 6.0的注册表

Windows Embedded CE 6.0中的注册表存放了与应用程序、驱动程序、用户自定义设置等相关的数据。同时，在Windows Embedded CE 6.0系统中，有两种方式来实现注册表，它们分别是：

- 基于RAM的注册表（RAM-Based Heap File）。
- 基于HIVE的注册表（Hive-Based Registry）。

在这里需要特别指出的是，因为基于RAM的注册表会在系统冷启动以后丢失数据，因此通常情况下，系统默认使用基于Hive的注册表来防止数据的丢失，但是使用基于RAM的注册表则可以大大地提高系统的效率。为了解决执行效率和数据丢失之间的矛盾，引入了RegCopyFile()函数和RegRestoreFile()函数来对注册表进行备份。其中RegCopyFile()函数的函数原型为：

```
BOOL RegCopyFile
{
    LPCWSTR lpszFile
};
```

RegRestoreFile()函数的函数原型为：

```
BOOL RegRestoreFile
{
LPCWSTR lpszFile
};
```

参数lpszFile为一个指向注册表备份文件的指针。

正是因为注册表具有上述特点，因此要想使驱动程序在设备启动的时候就会自动地加载，就

需要修改系统的注册表项。可以说，在Windows Embedded CE 6.0中，流接口驱动程序和注册表有着非常紧密的联系。

在早期的MS-DOS中，并不存在注册表这个概念，所有的配置文件都保存在扩展名为INI或SYS的文件里。随着Windows演变成了一个操作系统，而不仅仅是一个图形外壳，以及Windows 95相比原先的MS-DOS更加庞大和复杂，微软引入了注册表来统一管理Windows所有的配置信息以及加载和启动选项。

Windows CE从Windows 95及以后系列继承了注册表的机制，修改WinCE的注册表，就可以定制WinCE的各项参数。

Windows CE并不真正地包含一个如同Windows 2000/XP那样完全的、复杂的注册表数据库，但是类似于这些系统，WinCE的注册表同样担负着重要的配置信息保存的任务，记录着系统中若干重要的配置信息。

17.5.1　Windows Embedded CE 6.0注册表的结构

注册表实际上可以被看作是一个存储系统资源的数据库，里面通常包含了应用程序和Windows系统本身的一些配置信息。尽管Windows CE中的注册表和其他的Windows系统有一点类似，但是在一些函数及其使用上仍旧有很大的不同。在所有的Windows系统中，注册表都是由若干个键及其键值所组成。这一点类似文件系统，键相当于目录，而值则相当于文件。

在Windows Embedded CE 6.0中，系统注册表有以下四个根键：

- HKEY_LOCAL_MACHINE。
- HKEY_CLASSES_ROOT。
- HKEY_CURRENT_USER。
- HKEY_USER。

上面四个根键各自存储的信息为：

- HKEY_LOCAL_MACHINE是用来存储硬件和驱动程序的配置信息。HKEY_LOCAL_MACHINE根键作为Windows CE驱动开发最重要的根键，每个驱动都要在该根键下保存相应的信息，以便被Windows CE系统加载和调用。
- HKEY_CLASSES_ROOT用来存储文件类型和OLE配置信息。
- HKEY_CURRENT_USER用来存储用户配置信息。
- HKEY_USER则用来存储所有用户的共用数据。

图17-3为Windows CE中的注册表。

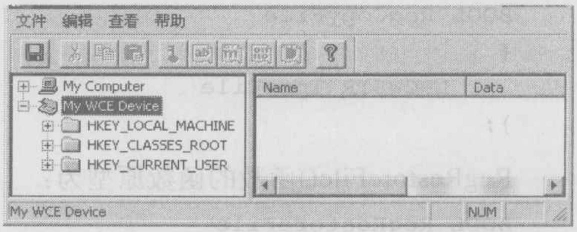


图17-3　Windows CE中的注册表

在这里有一点需要提及的是，文件类型由扩展名确定，当双击一种类型的文件，Windows就会在这个根键下自动寻找与之相关的信息，找出与之关联的应用程序，并使用此应用程序打开该类型的文件。

17.5.2　Windows Embedded CE 6.0的常用键值类型

在Windows Embedded CE 6.0中，常用的键值类型有以下几种。

• REG_BINARY：二进制数据，允许读写。注册表中所有二进制数据必须是偶数个字节的数，即可以表示成为多个十六进制，如01 AB 00 00 30。

• REG_DWORD：32位无符号整型。可以用十六进制也可用十进制来表示，如0xABCDEF、0x00000001。

• REG_SZ：字符串类型，如“Hello WinCE！”。

在Windows Embedded CE 6.0中，注册表采用层次结构来存储，在注册表的最顶端是三个根键，每个根键下存储一类配置信息。注册表的每个键可以不带设置值或带有多个设置值。在同一子键内，值不能相同，但是在不同子键下可以存在同名的值。

17.5.3 流驱动程序用到注册表设置

我们知道，在Windows CE中，流接口函数是由设备管理器进行调用的。因此流接口驱动程序用到的注册表设置实际上也就是设备管理器用到的注册表设置。

在Windows CE中，设备管理器使用存储在HKEY_LOCAL_MACHINE\Drivers\键中的注册键。图17-4为Pocket PC设备中的注册表信息。

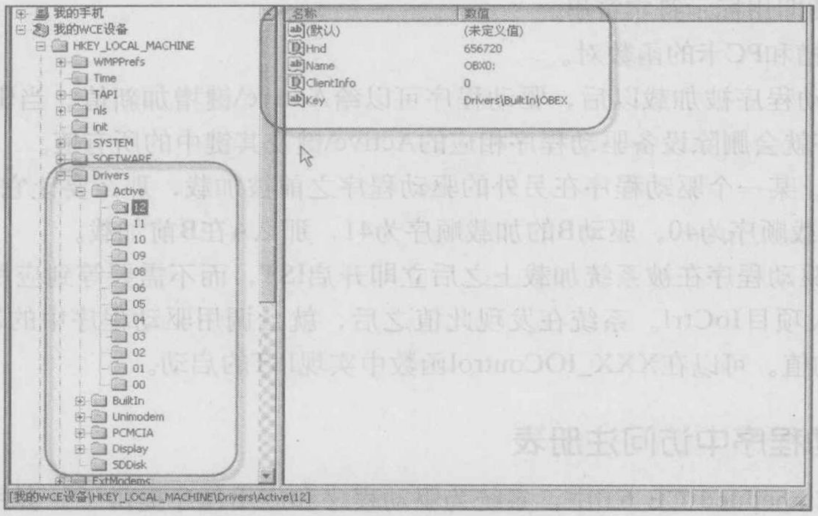


图17-4 Pocket PC设备中的注册表信息

在HKEY_LOCAL_MACHINE\Drivers\中，它包含的主要子键有：

1）Active\

Active\包含由设备管理器加载的，用于跟踪当前活动驱动程序的子键。在这里需要指出的是，设备驱动程序不要修改Active\键的内容，也不要依赖于Active\键中任何特殊值的存在。当加载一个流接口驱动程序的时候，设备管理程序会把它的Active\键的路径以XXX_Init函数的dwContext参数形式传递给设备驱动程序。设备驱动程序可以依赖于下列值的存在：

• Hnd：从RegisterDevice函数来的设备处理。

• Name：设备文件名。

• Key：HKEY_LOCAL_MACHINE中设备驱动程序键的注册表路径。

2）Builtin\

Builtin\键包含由设备制造商提供的管理流接口驱动程序DLL的子键。通过流接口驱动程序模

型，可以实现一些内部设备的驱动程序。在该子键下包含的项目主要有：

• Index：当前流驱动所对应的设备缩写，由三个大写字母构成。与驱动程序中的缩写一致。
• Prefix：该设备的索引号，从0到9都可以。
• Dll：设备驱动程序编译后的动态链接库的名称。
• Order：设备驱动程序的加载顺序。

3）PCMCIA\

PCMCIA\键含有与PC卡以及流接口驱动程序有关的子键。在这个子键中，最重要的信息就是PC卡的即插即用标示符。

4）Detect\

Detect\子键包含了已经编号的条目项。这些项都是DLL名称和侦测函数的列表。这些函数可以为无即插即用标示符的PC卡或者未知的即插即用设备提供一个通用的流接口驱动程序。

在这里还有几个问题，需要特别加以注意：

1）作为PC卡的设备驱动程序，其Active\键下除了上面提到的几个键值以外，还有其他的两个键值：

• PnpId：即插即用标示符字符串。
• Sckt：当前槽和PC卡的函数对。

2）当一个驱动程序被加载以后，驱动程序可以给Active\键增加新值。当驱动程序卸载的时候，设备管理程序就会删除设备驱动程序相应的Active\键及其键中的所有值。

3）如果打算让某一个驱动程序在另外的驱动程序之前被加载，那么要让它的加载顺序领先。例如，驱动A的加载顺序为40，驱动B的加载顺序为41，那么A在B前加载。

4）如果希望驱动程序在被系统加载上之后立即开启IST，而不需要等到应用程序调用其接口来加载，可以加入项目IoCtrl。系统在发现此值之后，就会调用驱动程序中的XXX_IOControl，调用参数为该项的值。可以在XXX_IOControl函数中实现IST的启动。

17.5.4 在驱动程序中访问注册表

在Windows Embedded CE 6.0中，系统为驱动程序和应用程序提供了一套用于访问注册表的函数，它们包括：

• RegOpenKeyEx：打开一个注册表键。
• RegCloseKey：关闭指定的注册表键。
• RegQueryValueEx：查询指定注册表项的值。
• RegSetValueEx：设置指定的注册表项的值。
• RegCreateKeyEx：生成一个新的注册表键。
• RegDeleteKey：删除指定的注册表键。

其中RegOpenKeyEx函数的函数原型为：

```
LONG RegOpenKeyEx( HKEY hKey,
                   LPCWSTR lpSubKey,
                   DWORD ulOptions,
                   REGSAM samDesired,
                   PHKEY phkResult );
```

各个参数的含义为：

- hKey：参数hKey为指向当前打开键值的句柄。
- lpSubKey：参数lpSubKey为指向打开子键的句柄。
- ulOptions：参数ulOptions需要被设置成0。
- samDesired：参数samDesired也需要被设置成0。
- phkResult：参数phkResult为指向接收键值的变量的指针。

RegCloseKey函数的函数原型为：

```
LONG RegCloseKey( HKEY hKey );
```

里面的唯一参数为需要关闭的注册表键的句柄。

RegQueryValueEx函数的函数原型为：

```
LONG RegQueryValueEx(     HKEY hKey,
                          LPCWSTR lpValueName,
                          LPDWORD lpReserved,
                          LPDWORD lpType,
                          LPBYTE lpData,
                          LPDWORD lpcbData );
```

各个参数的含义为：

- hKey：参数hKey为指向当前打开键值的句柄。
- lpValueName：参数lpValueName为需要查询的键值。
- lpReserved：参数lpReserved需要被设置成NULL。
- lpType：参数lpType为指向接收所查询健值类型的变量的指针。
- lpData：参数lpData为指向接收键值的缓冲区的指针。
- lpcbData：参数lpcbData为缓冲区的大小。

RegSetValueEx函数的函数原型为：

```
LONG RegSetValueEx( HKEY hKey,
                    LPCWSTR lpValueName,
                    DWORD Reserved,
                    DWORD dwType,
                    const BYTE *lpData,
                    DWORD cbData );
```

各个参数的含义为：

- hKey：参数hKey为指向当前打开键值的句柄。
- lpValueName：参数lpValueName为需要设置的键值的名称。
- Reserved：参数Reserved需要被设置成 0 。
- dwType：参数dwType为键值的类型。
- lpData：参数lpData为存储键值的缓冲区。
- cbData：参数cbData为缓冲区的大小。

RegCreateKeyEx函数的函数原型为：

```
LONG RegCreateKeyEx(  HKEY hKey,
```

```
                    LPCWSTR lpSubKey,
                    DWORD Reserved,
                    LPWSTR lpClass,
                    DWORD dwOptions,
                    REGSAM samDesired,
                    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                    PHKEY phkResult,
                    LPDWORD lpdwDisposition );
```

各个参数的含义为：

- hKey：参数hKey为指向当前打开键值的句柄。
- lpSubKey：参数lpSubKey为需要创建的子键的名称。
- Reserved：参数Reserved需要被设置成0。
- lpClass：参数lpClass为子键对象的类型。
- dwOptions：参数dwOptions需要被设置成0。
- samDesired：参数samDesired需要被设置成0。
- lpSecurityAttributes：参数lpSecurityAttributes需要被设置成NULL。
- phkResult：参数phkResult为接收所创建子键的变量。
- lpdwDisposition：参数lpdwDisposition为REG_CREATED_NEW_KEY（创建了不存在的键值）或REG_OPENED_EXISTING_KEY：打开一个已经存在的注册表键。

RegDeleteKey函数的函数原型为：

```
LONG RegDeleteKey( HKEY hKey,
                   LPCWSTR lpSubKey );
```

各个参数的含义为：

- hKey：参数hKey为指向当前打开键值的句柄。
- lpSubKey：参数lpSubKey为需要删除的子键的名称。

17.6 流接口驱动程序的加载和卸载

在Windows Embedded CE 6.0中，流接口驱动程序的加载可以有三种方法，其中两种涉及设备管理器，一种涉及应有程序。相应的流接口驱动程序的卸载也就存在两种方式，分别针对设备管理器和应用程序。

下面首先来看一下流接口驱动程序的加载。

17.6.1 流接口驱动程序的加载

在Windows CE 6.0中，第一种加载流接口驱动程序的方式是在启动的时候进行加载。也就是说，当Windows CE 6.0平台启动的时候，设备管理程序就随即启动。设备管理程序会读取HKEY_LOCAL_MACHINE\Drivers\BuiltIn下的内容并加载已经列出的流接口驱动程序。例如，在很多的Windows CE 6.0系统中，设备管理器就是通过这个机制进行内部串口驱动程序(Serial.dll）的加载的。

在Windows Embedded CE 6.0中，第二种加载流接口驱动程序的方式就是在设备管理器自动

检测外围设备与系统连接的时候进行的。例如，在前面经常提到的PC卡自动检测等。当把PC卡插入到PC卡插槽的时候，设备管理程序就会调用插槽的驱动程序寻找即插即用设备。然后设备管理程序就会检查HKEY_LOCAL_MACHINE\Drivers\PCMCIA键来得到与即插即用设备相匹配的子键。当该子键存在的时候，子键就会加载键值列表里面的驱动程序；如果子键不存在，该设备管理程序就会调用HKEY_LOCAL_MACHINE\Drivers\PCMCIA\Detect里面的所列的所有侦测函数。当某个函数返回一个值的时候，就表明它在处理我们插入的PC卡，设备管理程序就会加载并初始化这个流接口驱动程序。

在这里需要指出的是，在上面提到的两种加载方式中，设备管理程序会调用RegisterDevice函数，并在HKEY_LOCAL_MACHINE\Drivers\Active键中生成一个已经编号的子键来跟踪这个驱动程序。RegisterDevice函数还会把流接口驱动程序锁入到RAM中，以防止该驱动程序被置换出。也防止由于页面调度活动而导致的对流接口驱动程序操作速度的减慢。

上面介绍的两种流接口驱动程序的加载方式都和设备管理器相关，下面再来看一下与驱动程序相关的加载方式。

当系统的设备管理程序不能自动监测到外围设备的时候（如在Windows CE.Net中系统不能自动监测串行设备和串行设备的连接），就需要相应的应用程序来加载这个驱动。使用应用程序加载流接口驱动程序时的加载顺序为：

（1）需要启动应用程序。

（2）应用程序需要确定驱动程序还未被加载，或者通过检测文件系统，或者在应用程序试图和外接的设备进行通信的时候，从CreateFile函数中返回了一个失败值。

（3）应用程序会调用RegisterDevice函数进行相应的流接口驱动程序的加载。

（4）应用程序开始其他正常的工作。

在驱动程序加载后，会在HKEY_LOCAL_MACHINE\Drivers\Active子键下创建一个新的子键，子键的名称为数字，且逐个递增。子键内的项目主要包括：

- Name：设备名。
- Key：注册设备的路径。
- PnpId：即插即用字符串，只为PC卡驱动准备。
- Sckt：当前插口和PC卡的函数对。

17.6.2 流接口驱动程序的卸载

在Windows Embedded CE 6.0中，可以根据设备管理器和应用程序两种加载方式（和设备管理器相关的加载方式中又分为两种）来决定流接口驱动程序的卸载方式。

- 设备管理程序加载：当设备管理程序加载了流接口驱动程序以后，设备管理就会检测该驱动程序是否已经和Windows CE平台断开连接，如插入的SD卡是否已经从插槽中拔出。此时设备管理程序会从HKEY_LOCAL_MACHINE\Drivers\Active中删除驱动程序项，并调用DeregisterDevice函数从文件系统中删除外接设备的设备文件名称。同时还会通知FreeLibrary函数卸载DLL。
- 应用程序加载：当流接口驱动程序是通过应用程序进行加载的，应用程序就必须在驱动程序存在前通过调用DeregisterDevice函数来卸载这个DLL。

第18章 Windows Embedded CE 6.0中的总线

18.1 总线概述

Windows Embedded CE 6.0中用到的总线驱动主要负责决定加载哪一个驱动程序以及以什么顺序加载总线上的设备。Windows Embedded CE 6.0中的驱动主要包含以下几个内容：

- 控制硬件设备。
- 协调各设备驱动对总线的访问。
- 完成设备驱动和具体物理设备之间的数据交换。
- 提供统一的上层驱动接口。

在详细了解Windows Embedded CE 6.0系统各种类型的总线技术以前，首先来看一下总线相关的知识。

总线是计算机系统中，各个部件之间进行数据传输以及进行地址和控制信息传输的公共通路。通俗地说，总线就是计算机系统中各种信号线的集合。

在一个计算机系统中，任何一个微处理器都需要与一定数量的部件和外围设备相连接，如果将各个部件和每一种外围设备都与CPU直接相连接，那么连接将会非常的复杂，有的甚至是难以实现的。因此为了简化硬件电路的设计、简化系统的结构，常用一组线路并配置适当的接口电路，与各个部件和外围设备相连接。这些共用的连接线路就被称为总线。

总线技术最早出现在1970年。在这一年，美国DEC公司推出了采用Unibus总线技术的PDP11/20小型计算机。后来随着技术的不断进步，以及计算机市场的不断扩大和应用领域的日益广泛，陆续出现了各种标准的、非标准的总线。总线技术之所以发展得如此迅速，是因为总线结构在系统设计、生产、使用和维护上有着诸多的便利和优越性。主要表现在以下几点：

- 便于系统采用结构化的设计方式，有效地简化了设计流程。
- 所采用的总线标准得到了众多厂商的支持，有利于生产出兼容性较高的硬件产品和软件产品。
- 由于其采用了模块化结构，因此系统利于升级和扩充。
- 同样由于其使用的模块化结构，便于系统的维护，有效地降低了维护成本。

实际上从PC诞生以来，就一直采用着总线结构方式。总线技术的先进性对于提高整个计算机系统的性能有着十分重要的作用。因此随着近年来PC的迅速发展和普及，总线结构也在不断地发生着变化。

提到计算机中的总线，就不能不提到接口技术。接口技术在计算机中被广泛地运用同样是出于模块化设计的考虑，便于各个生产厂商生产出兼容性较高的外围硬件产品和软件产品。通常情况下，接口一般是指计算机主板和某类外设之间的适配电路，其主要功能就是解决主板和外设之间在电压、信号、传输速率等方面的匹配问题。因此不同类型的设备需要不同的接口。例如，常用的USB接口、鼠标、键盘采用的PS2接口，数码摄像机采用的1394接口等。由于接口可以连接

不同的外围设备，因此也把它们统称为外设总线。

18.2 总线的分类和主要参数

在一个计算机系统中，有各式各样的总线。这些总线可以从不同的层次和角度进行划分。下面就来看一下总线的分类方式。

18.2.1 总线的分类

按照相对于CPU的位置，总线可以分为片内总线和片外总线。片内总线是指在CPU内部，寄存器之间或者算数逻辑器件与控制部件之间进行数据传输所采用的总线。相对应地，片外总线是指CPU与内存、输入/输出设备之间进行通信所采用的总线。通常所说的总线主要是指片外总线。

如果按照功能进行划分，总线又可以被分为：

- 地址总线：用来传送地址信息。
- 数据总线：用来传送数据信息。
- 控制总线：用来传送控制信号。

在这里需要特别指出的是，通常情况下所说的总线都包含上述三个组成部分。如果按照总线的层次结构进行划分，总线又可以被分为：

- CPU总线：其主要功能是连接CPU和控制芯片。
- 存储总线：其主要功能是连接存储控制器和DRAM。
- 系统总线：其主要功能是与扩展插槽上的扩展卡相连接，如经常会在各种嵌入式设备及嵌入式实验平台上看到的各种扩展模块插槽就属于系统总线。
- 外部总线：其主要功能是连接外设的控制芯片，如主机板上的I/O控制器和键盘控制器。

如果按照总线在PC中的位置，又可以把总线分为：

- 机内总线。
- 机外总线。

在前面提到的各种分类方式中的总线都属于机内总线。而机外总线实际上是一种外围设备的接口标准，如经常会看到的IDE、USB、SCSI等。

在Windows桌面系统中，包括本书介绍的Windows CE系统，图形用户界面已经大行其道。这就要求系统有较强的图形图像绘制能力和I/O处理能力，因此也就对总线结构提出了更大的要求。在这种情况下，又出现了一种被称为局部总线的总线结构。所谓的局部总线，就是在ISA总线和CPU总线之间增加的一级总线或管理层。通过这种方式，可以将一些图形卡、硬盘控制器等高速外设从ISA总线上卸载下来，而通过局部总线把它们直接挂载到CPU总线上，达到了与高速的CPU总线之间匹配。可以说，局部总线的提出打破了传统PC体系结构中数据I/O的瓶颈，极大地发挥了CPU的潜能。

局部总线可以分为以下三种：

- 专用局部总线。
- VL总线。
- PCI总线。

这里提到的专用局部总线是类似NEC、HP这样的大公司为了开发自己系统所需而提出的专用

总线，主要用于图形处理、网络传输等。因为是专用总线，因此和其他的总线结构不同，不被广大的兼容机所采用。

VL总线是适用于486的一种过渡标准，已经被淘汰。在586以上的机器中主要采用PCI总线。在Windows CE系统中，也广泛采用PCI总线结构。在本章的后面会对Windows CE.Net中的PCI总线作详细的介绍。

18.2.2 总线的主要参数

总线参数主要有：

- 总线带宽。
- 总线位宽。
- 总线的工作时钟频率。

其中的总线带宽是指一定时间内总线上可以传输的数据量，也就是常说的每秒传输多少兆字节的最大稳态数据传输率。与它紧密相关的两个概念就是总线的位宽和总线的工作时钟频率。

总线位宽主要指的是总线能够同时传送的数据位数。也就是常说的32位、62位等。总线的位宽越宽则表示总线每秒的数据传输率越大。而在PC系统中，总线的工作频率越高，则表示总线的工作速度越快。总线的工作时钟频率是以MHz为单位的。

在实际的运用中，只有配合地提高总线的位宽和总线的工作时钟频率，才能比较有效地提高总线的带宽。如果单方面地提高其中之一，很难达到实际的效果，并且很容易达到各自的限制。

18.3 内部集成总线

内部集成总线（IIC）的英文全称是Inter Integrated Circuit，它是由PHILIPS公司开发的二线式串行总线。主要作用是连接微控制器以及外围设备。内部集成总线产生于20世纪80年代，最早是为音频和视频设备开发的。随着技术的不断进步和应用的日益广泛，内部集成总线近年来被广泛地应用于微电子通信控制领域，已经成为被广泛采用的一种新型总线标准。实际上，内部集成总线是同步通信的一种特殊形式。它具有接口线少，控制方式简单，通信速率高，器件封装形式小等诸多优点。在主从通信方式中，可以有多个内部集成总线器件连接到内部集成总线上，通过地址来识别通信的对象。

18.3.1 内部集成总线的特点

内部集成总线具有下面几个最主要的优点：

- 简单性。
- 有效性。
- 支持多主控（multimastering）。

因为相应的接口直接接在组件之上，因此内部集成总线所占用的空间相对较小，这样就减少了电路板的空间和芯片引脚的数量，进而降低了连接的成本。

所谓的支持多主控主要是指启动任何可以进行发送和接收的设备都可以成为主总线。一个主控可以控制信号的传输和时钟的频率。在这里需要特别指出的是，在任何时刻，只能有一个主控存在。同时，内部集成总线设备只是在需要作为主机及发送数据的时候才会通过总线仲裁使自己

成为主机，其他时刻都是作为从机，等待其他主机呼叫本机地址。

内部集成总线驱动程序要完成的工作主要包括：

- 配置IIC总线。
- 处理IIC控制器中断和各种协议信号（开始、停止）。
- 完成IIC传输协议所要求的数据写入和读出。
- 如果针对一种传输模式，则仅仅是完成此模式；如果同时处理一种以上的传输模式，则要在这些模式之间协调。

内部集成总线是由数据线SDA和时钟SCL构成的串行总线，具有发送和接收数据的功能。它在数据的传输过程中可以传送三种类型的信号：

- 开始信号。
- 结束信号。
- 应答信号。

其中开始信号是指时钟SCL为高电平时，SDA由高电平向低电平进行跳变，开始传送数据；结束信号是指时钟SCL为低电平时，SDA由低电平向高电平进行跳变，结束传送数据；而应答信号则是指接收数据的IC在接受到8bit的数据以后，向发送数据的IC发出特定的低电平脉冲，表示已经收到数据。在CPU向受控单元发出一个信号以后，就会等待受控单元发出的应答信号。CPU在接收到这个应答信号以后，就会根据实际的情况作出是否需要继续传递信号的判断。如果CPU没有收到应答信号，则判断为受控单元出现了故障。

18.3.2 传输方式

在嵌入式系统中，如本书介绍的Windows Embedded CE 6.0中，内部集成总线（IIC）一共支持4种数据传输方式，它们分别为：

- 主机发送。
- 主机接收。
- 从机发送。
- 从机接收。

一个设备处于主机发送模式时，必定有另一个设备处于从机接收模式；同理，当一个设备处于主机接收模式时，必定有另一个设备处于从机发送模式。

由于IIC总线属于半双工传输总线，因此在同一时刻，只能由一个方向的数据传送。针对IIC的四种传输模式，总线上的数据状态只能有两种可能：

- 由主机向从机发送。
- 由从机向主机发送。

图18-1展示了这两种情况。

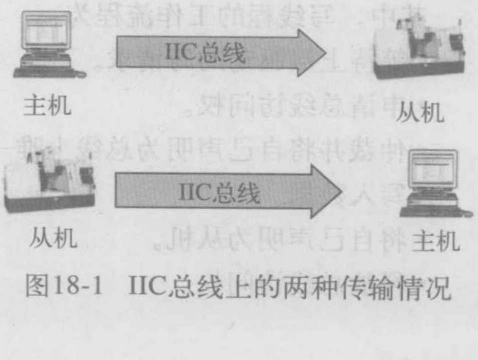


图18-1 IIC总线上的两种传输情况

刚才提到过，内部集成总线设备只是在需要作为主机及发送数据的时候才会通过总线仲裁使自己成为主机。仲裁的过程大致为：

1）开始一次数据传输，主机首先发送地址最高位。

2）检测总线是否是期望的状态。

3）如果是，则发送下一位，继续检测；如果不是，则放弃当前总线。

4）如果所有的地址位都已发送完毕，且没有出错，则仲裁成功。

18.3.3 内部集成总线与硬件设备之间的数据交换

在总线驱动程序中需要设计两个线程：读线程和写线程。其中读线程用于在从机接收状态下从总线上读取数据，写线程用于在主机发送状态下向总线写入数据。读线程和写线程交替控制总线，就可以完成驱动程序和IIC总线的数据交换。

从总线上得到的数据可能是由各种设备或主机发送来的，总线驱动程序的缓冲区就像是这些来源大小不同的数据的集散地，为了能够完成数据的处理，必须将这些数据按照其指定的位置分发出去。

IIC总线上的数据由字节组合成包，在每一个数据包的开始都会加上一个START信号，相应的STOP信号表明数据包的结束。通过探测START和STOP信号，就可以确定数据包的确切大小。同时，数据包是连续的，一个数据包有一个明确的用于接收它的设备驱动程序。IIC总线上的数据流由若干个数据包组成，每个数据包可能目的地并不相同。

为了合理地将数据包分发到设备驱动程序，可以要求在数据包的第一个字节注明自己的去向，只要判断此字节的值，就知道要分发到哪里。在总线驱动程序中，专门设定一个分发函数，每当总线上收到了数据，就将这个字节发送给分发函数，分发函数根据情况将其发送给各个设备驱动程序。

图18-2展示了IIC总线获得并处理数据的过程。

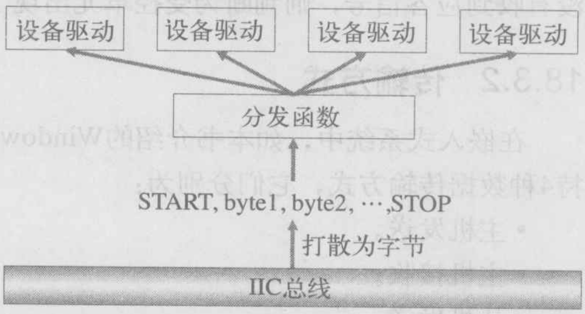


图18-2 IIC总线获得并处理数据的过程

总线驱动程序需要对硬件进行全面的控制，包括读数据和写数据。IIC总线驱动程序必须具备向硬件写入数据的功能。

向总线写入数据必须首先确保本机是总线上唯一的一个主机。为此要进行一个所谓仲裁的过程。如果仲裁成功，驱动程序会收到一个中断，如果仲裁不成功，同样也会收到一个中断。仲裁成功后，本机成为总线上唯一的一个主机，这时可以发出START信号和目标从机地址，开始一个数据包的传送。

当IIC总线空闲的时候，本机一定要处于从机接收模式，这就决定了写线程不能长时间地占用总线等待上层驱动程序的写请求，只能是一直在等待上层驱动程序的写请求，随后申请总线的使用权。

其中，写线程的工作流程为：

- 等待上层驱动的写请求。
- 申请总线访问权。
- 仲裁并将自己声明为总线上唯一主机。
- 写入数据。
- 将自己声明为从机。
- 释放总线访问权。

• 继续等待上层驱动的写请求。

作为总线驱动程序，一个重要的任务就是和设备驱动程序进行数据交换，设备驱动程序首先向总线驱动程序发出读写请求，随后总线驱动做出相应的动作。

每个设备驱动程序对应一个缓冲区，这个缓冲区对于设备驱动和总线驱动都是可见的。为了达到这一目的，使用内存文件映射。WinCE允许将一个文件映射到内存中，以便快速访问以及进程间共享，这种机制同时也允许开辟一块内存区作为数据交换时使用，这块内存区并不映射到任何文件。内存文件映射位于全局堆中，既可以被内核态的进程访问，也可以被用户态的进程访问。利用这一性质将这块内存作为设备驱动程序和总线驱动程序进行数据交换的缓冲区。

设备驱动对总线的请求包括读和写两方面，如果只有一个缓冲区，那么只有当前一个请求被处理后，后一个请求才能被放入缓冲区，如果对于读或者写分别有一个缓冲区，情况就大不相同了。设备驱动对总线的读操作可以看作是设备驱动程序从缓冲区中读数据，而总线驱动向缓冲区里面写数据；设备驱动对总线的写操作可以看作是总线驱动从缓冲区中读数据，而设备驱动向缓冲区中写数据。

图18-3展示了对总线的读请求，图18-4展示了对总线的写请求。

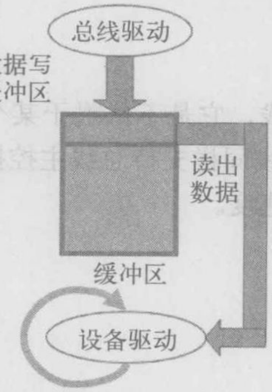


图18-3　对总线的读请求

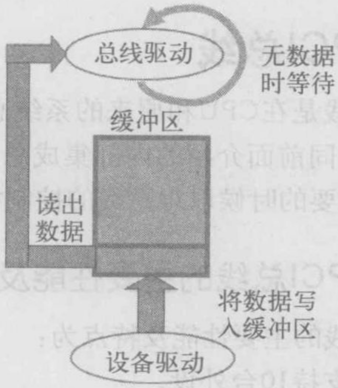


图18-4　对总线的写请求

从上面介绍的内容中可以看到，为了实现总线驱动和设备驱动之间的交互，要准备两个缓冲区，并且设计一套总线驱动和设备驱动之间同步的方法。

缓冲区在同一时刻只能被一个进程所访问，缓冲区的第一个字节表明此缓冲区内有效数据个数，如果任何一方读取此缓冲区的数据，则一次将所有有效数据读走，并将第一个字节清零，表示已不存在有效数据；相应地，任何一方向缓冲区内写数据时，要么接着原先的数据写入，要么此时缓冲区内已无有效数据，直接将内容从头写起，无论如何都要改写第一个字节，表明此时有效数据的个数已经发生了变化。

针对读、写两种操作，可以设置两对事件，一对事件对应着读请求，一对事件对应着写请求。使用一对事件的目的是一个事件用于通知，一个事件用于响应。

例如，设备驱动要对缓冲区进行读操作，首先触发读操作的通知事件，如果此时总线驱动没有对读缓冲区进行操作，那么总线驱动将触发读响应事件，否则一直等到对缓冲区的操作完成才触发读响应事件。如果总线驱动要对读缓冲区进行写操作，那么首先触发写通知事件，如果此时设备驱动已完成对缓冲区的操作则触发写响应时间，否则直到操作完成才触发响应事件，总线驱

动只有在接收到响应时间后才会对缓冲区进行写操作。

图18-5为总线驱动和设备驱动之间同步的示意图。

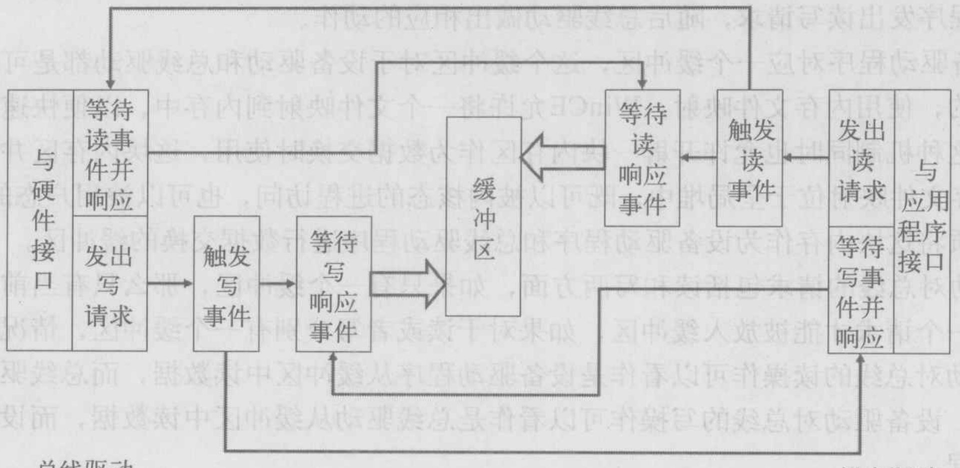


图18-5 总线驱动和设备驱动之间同步的示意图

18.4 PCI总线

PCI总线是在CPU和原来的系统总线之间插入的一级总线，它是不依附于某个具体处理器的局部总线。同前面介绍的内部集成总线（IIC）一样，PCI总线同样支持总线主控技术，即允许智能设备在需要的时候取得总线的控制权，以加速数据传输的速度。

18.4.1 PCI总线的主要性能及特点

PCI总线的主要性能及特点为：

- 最多支持10台外设。
- 最大数据传输速率为133MB/s。
- 与CPU及时钟频率无关。
- 总线时钟频率为33.3MHz/66MHz。
- 总线宽度为32位（5V）/64位（3.3V）。
- 适合与Intel的CPU协同工作。
- 具有隐含的中央仲裁系统。
- 能够自动识别外设。
- 支持64位寻址。
- 提供地址和数据的奇偶校验。
- 由于采用了多路复用方式，因此较少了引脚数。
- 完全的多总线主控能力。
- 可以转换5V和3.3V的信号环境。

在计算机技术、控制技术迅速发展的过程中，PCI总线凭借其优越的电气性能，得到了工业界一致的认可。在此基础上，利用PCI总线的特性，同时结合不同的机械结构，产生了各种各样的总线。

18.4.2 PCI总线的外设自动识别

这里面提到的外设自动识别实际上就是经常提的即插即用。所谓即插即用就是指当板卡插入到系统的时候，系统会自动地对设备所需资源进行分配，如自动配置基地址等。同时还会自动地寻找设备驱动程序。PCI总线之所以具备这样的功能，是因为在PCI板卡中有一组用来存放基地址、内存地址以及中断信息的寄存器，我们把它称为“配置空间”。

当设备加电的时候，板卡会从ROM里面读取固定的值放在寄存器中。操作系统就会根据这个信息对资源进行分配。在资源分配成功以后，就会在相应的寄存器内添加上内存的起始地址等信息。这样就实现了PCI总线对外设的自动识别，而不用再手工地配置相关资源了。

18.4.3 Windows CE 6.0中PCI总线驱动的作用

在Windows CE 6.0中，PCI总线驱动主要负责列举PCI总线，并加载在PCI总线上找到的设备驱动程序。这里提到的PCI总线列举是指通过查找PCI总线设备，并为PCI设备配置和分配系统资源的一个过程。配置和分配的系统资源主要包括：

- 内存基地址。
- I/O基地址。
- IRQ号。

比如在Windows CE.Net中，PCI总线驱动是由总线枚举器进行加载的。可以在WinCE的安装目录中看到PCI驱动的源代码。图18-6展示了PCI驱动程序的路径及源代码文件夹。

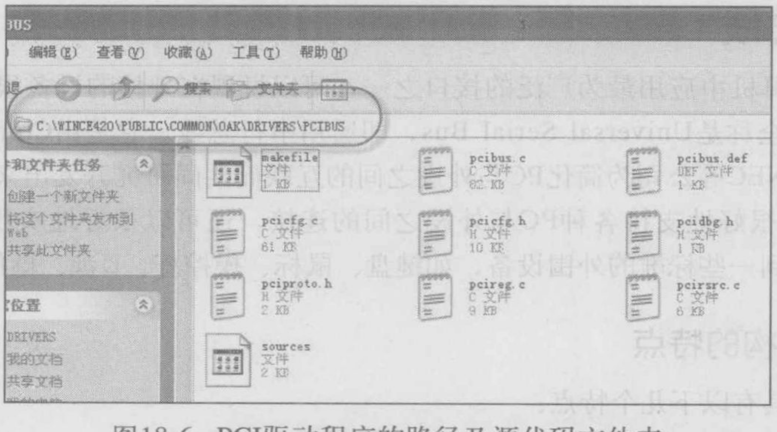


图18-6　PCI驱动程序的路径及源代码文件夹

在Windows CE中，PCI总线是可以配置的。它可以为连接到总线上的PCI设备灵活地配置所需的资源。在设备加载的过程中，所有使用固定资源的设备首先被加载，最后加载的才是PCIBus.dll。此时，它会从系统剩下的可用资源中再次对资源进行分配。

18.4.4 PCIBus.dll对PCI驱动的加载

在Windows Embedded CE 6.0中，PCI总线驱动程序被实现为一个名为PCIBus.dll的动态链接库，并由总线枚举器进行加载。PCIBus.dll加载PCI驱动程序的方式为：

1）PCIBus.dll首先扫描PCI总线上的设备。

2）对于扫描到的设备，在注册表PCI\Template或者PCI\Instance键下寻找与设备相匹配的注

册表项。

3）当找到一个相匹配的注册表项的时候，PCIBus.dll就会检查ConfigEntry的键值。如果该键值存在，它就加载PCI设备驱动程序的DLL，并调用由ConfigEntry描述的配置函数来帮助分配和配置设备资源。

4）PCIBus.dll再次扫描PCI总线，当再次发现一个PCI设备的时候，就会再次检查PCI\Template或者PCI\Instance。

5）如果在PCI\Instance下找到了一个相匹配的注册表项，就表示这个设备已经被配置并已经被系统所识别。

6）如果在PCI\Template下找到了一个相匹配的注册表项，那么PCI\Template下的所有信息都会被复制到PCI\Instance下。

7）PCIBus.dll通过扫描PCI总线上的设备和列举PCI设备所需的资源来列举PCI总线。

8）当PCI总线被配置以后，PCIBus.dll就会扫描在PCI\Instance下的每一个子键，并逐个加载PCI设备驱动程序。

9）PCIBus.dll为每一个PCI设备分配资源。

10）PCIBus.dll再次扫描和匹配在PCI\Template下定义的设备，并在PCI\Instance下寻找严格的匹配。

11）如果匹配成功，该设备的一个注册表入口将会被加载到注册表[HKEY_LOCAL_MACHINE\Drivers\Active键下。

18.5 USB总线

USB是目前计算机中应用最为广泛的接口之一，可以说现在外接的设备很多都是采用USB接口的。USB的英文全称是Universal Serial Bus，即通用串行总线。它是由Compaq、DEC、IBM、Intel、Microsoft、NEC等公司为简化PC与外设之间的互联而共同研究开发出来的一种免费的标准化连接器。它可以很好地支持各种PC与外设之间的连接，还可以很好地实现数字多媒体集成。USB技术直接影响到一些标准的外围设备，如键盘、鼠标、操控杆、U盘、摄像头等。

18.5.1 USB结构的特点

USB总线结构具有以下几个特点：

- USB为所有的设备提供了单一的、易于定义的标准连接器类型。
- USB支持热插拔。
- USB在设备供电上提供了灵活性。
- USB使得多个外围设备能够同时跟主机进行通信。

所谓的USB总线结构的即插即用是指USB连接器可以将各种各样的外设I/O端口整合在一起，使之具有自动识别和配置的能力。在使用的时候，只需要把USB设备插入到PC的USB接口中，设备就可以自动地对其进行识别，并自动地调用相应的驱动程序。

USB总线结构的另外一个显著特点就是支持热插拔。所谓的热插拔就是指在使用的时候，不需要关闭计算机的电源，可以在机器正常运行的时候在设备上插入或者拔出一个USB设备。随着时间和技术的不断发展，USB技术已经成为了PC设备的标准技术。目前一些智能设备上也配置了

USB接口，或者mini USB接口。尤其是当USB技术标准由1.1升级至2.0以后，传输的速率也由12Mbit/s扩展到了480Mbit/s，更换介质后的连接距离也由5m增加到了近百米。这些都进一步增强了USB的性能，为USB技术更为广泛地应用奠定了基础。

18.5.2 USB设备的结构

USB采用4线电缆，其中两根是用来传送数据的串行通道，另外两根是为设备提供电源的。这也就是USB接口的鼠标、USB接口的移动硬盘（台式机硬盘除外）不需要外接电源的原因。在日常的设备中，较为常见的有以下两种USB接口：

- A类接口。
- B类接口。

图18-7为A类接口的图示说明，图18-8为B类接口的图示说明。

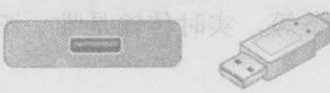


图18-7 A类USB接口

图18-8 B类USB接口

例如，打印机USB连接在一般情况下，连接打印机端的为B类接口，连接PC端的为A类接口。

一般情况下，USB系统由三个基本组件组成：

- 主控制器驱动程序HCD。
- USB驱动程序USBD。
- 主机软件。

其中的主控制器驱动程序HCD可以把不同的主控制器设备映射到USB系统中。HCD与USB之间的接口被称为HCDI，特定的HCDI由支持不同主控制器的操作系统来定义。在一个系统中，通用主控制器驱动器UHCD处于软件接口的底层，用来控制和管理主控制器。系统软件的最高层通过UHCD的软件接口与主控制器进行通信。

USB驱动程序USBD在主控制器驱动器UHCD的驱动器之上，它提供了驱动器级别的接口，以满足现有设备驱动器设计的要求。USBD以I/O请求包的形式提供数据传输架构，它又通过特定管道Pips进行数据的传输。同时，USBD也是客户端设备的一个抽象，作为抽象的一部分，USBD拥有默认的管道。通过它可以访问所有的USB设备以及进行标准的USB控制操作。该默认管道描述了一条USBD和USB设备之间通信的逻辑通道。

提到了管道，就需要有一个通信的来源（相当于管道的起点），这个来源被称为端点。端点是一个USB设备中唯一可寻址的部分，是主机与设备之间通信的来源或目的。设备在建立连接前端点就是存在的，而管道是设备要通信的时候才搭建的桥梁。所以端点是地址而管道是通信时候临时建立的路径。

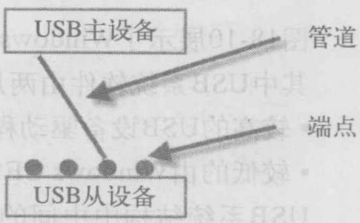


图18-9 USB设备端点和管道的拓扑结构

图18-9展示了USB设备端点和管道的拓扑结构。

主机软件则是指向设备驱动程序提供配置信息的。

18.5.3 USB设备的传输类型

USB设备主要有以下4种传输类型：

- 控制传输。
- 块传输。
- 中断传输。
- 实时传输。

其中控制传输用于主机读取设备的信息和设备的地址以及进行设备配置，所有USB必须支持控制传输，每个USB设备都必须将0端点设置成控制传输。块传输又叫批量传输，通常使用在速率不是很重要的场合，例如传送文件给打印机的时候。如果总线被其他必须保证速率的传输所占据时批量传输必须等待。设备不一定要支持批量传输。中断传输使用在设备必须定时接收主机或设备信息的场合。设备并不一定要支持中断传输。实时传输又叫同步传输，实时传输为了保证传输的速率没有错误检验，使用实时传输的设备包括音视频设备等，实时传输是唯一不会自动重传的传输类型。设备不一定要支持实时传输。

18.5.4 USB设备的软件设计

USB设备的软件设计主要包括下面两个部分：

- USB设备端的单片机软件。
- PC端的程序。

其中USB设备端的单片机软件主要是完成USB协议的处理与数据交换。在一些情况下还需要包含其他的应用程序功能，如A/D转换、MP3解码等。PC端的程序主要是由USB通信程序和用户服务程序两部分组成。其中用户服务程序是通过USB通信程序与系统USBDI（USB Device Interface）进行通信的。PC端程序的开发难度较大，程序开发人员不仅需要熟悉Windows CE的体系结构，熟练地使用各种开发工具，还需要对USB协议非常地了解。

对USB设备软件的调试基本上分为以下三个步骤：

（1）借助P C端的调试软件，将外设端的USB协议调通；

（2）用调试好的USB外设接口来开发、调试PC端的软件；

（3）加上USB外设的其他用户程序，对整个系统进行系统调试。

18.5.5 Windows CE 6.0下USB系统的组成

图18-10展示了Windows CE 6.0下的USB系统结构。

其中USB系统软件由两层组成：

- 较高的USB设备驱动程序层。
- 较低的由Windows CE实现的USB函数层。

USB系统结构中中间的一层则是由较高的通用串行总线驱动程序USBD模块和较低的控制器驱动程序HCD模块组成。HCD模块给USBD模块提供底层的支持，USBD实现高层的USBD接口函数。USB设备驱动程序使用USBD提供的接口函数和外围设备进行通信。

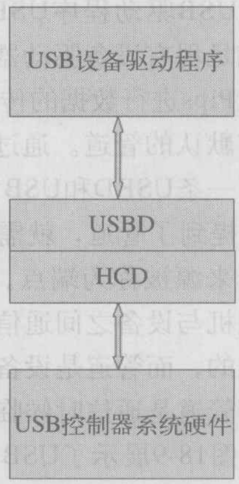


图18-10 Windows CE 6.0下的USB系统结构

18.5.6 Windows Embedded CE 6.0下USB设备驱动程序的编写

在Windows Embedded CE 6.0中，USB设备驱动程序的编写可以采取下面三种方式：

- 使用USBD模块提供的函数操作USB设备。
- 使用流接口函数。
- 使用现有的Windows CE应用程序编程接口（API）。

其中USBD的传输函数主要有：

- AbortTransfer。
- CloseTransfer。
- GetIsochResults。
- GetTransferStatus。
- IssuBulkTransfer。
- IssueControlTransfer。
- IssueInterruptTransfer。
- IssueIsochTransfer。
- IsTransferComplete。
- IssueVendorTransfer。

USBD和USB设备建立通信管道的函数主要有：

- AbortPipeTransfer。
- ClosePipe。
- IsDefaultPipeHalted。
- IsPipeHalted。
- OpenPipe。
- ResetDefaultPipe。
- ResetPipe。

USB针对在总线上的数据打包函数主要有：

- GetFrameLength。
- GetFrameNumber。
- ReleaseFrameLengthControl。
- SetFrameLength。
- TakeFrameLengthControl。

USBD与USB设备进行交互的函数主要有：

- FindInterface。
- GetDeviceInfo。
- GetUSBDVersion。
- LoadGenericIntefaceDriver。
- OpenClientRegistryKey。
- RegisterClientDriverId。
- RegisterClientSettings。

- RegisterNotificationRoutine。
- TranslateStringDescr。
- UnRegisterNotificationRoutine。

所有USB设备驱动程序必须在它们的Dll库中呈现一定的入口点从而与USBD模块进行适当的交互，Windows CE要求USB设备驱动所必须提供的入口点为：

- USBDeviceAttach。
- USBInstallDriver。
- USBUninstallDriver。

其中USBDeviceAttach的功能是：当USB设备连接到计算机上时，USBD模块就会调用此函数，这个函数主要用于初始化USB设备，取得USB设备信息，配置USB设备，并且申请必需的资源。该函数的函数原型为：

```
BOOL USBDeviceAttach( USB_HANDLE hDevice,
                      LPCUSB_FUNCS lpUsbFuncs,
                      LPCUSB_INTERFACE lpInterface,
                      LPCWSTR szUniqueDriverId,
                      LPBOOL fAcceptControl,
                      LPCUSB_DRIVER_SETTINGS lpDriverSettings,
                      DWORD dwUnused );
```

各个参数的含义为：

- hDevice：参数hDevice是USBD为此设备分配的句柄。
- lpUsbFuncs：参数lpUsbFuncs表示USBD函数指针。
- lpInterface：参数lpInterface为指向 USB_INTERFACE 结构的指针，包含必要的接口信息。
- szUniqueDriverId：参数szUniqueDriverId为设备ID描述字符串。
- fAcceptControl：参数fAcceptControl返回TRUE的时候，说明驱动接收该设备；当为FALSE的时候，表示驱动不接收该设备。
- lpDriverSettings：参数lpDriverSettings为指向 USB_DRIVER_SETTINGS结构的指针。
- dwUnused：参数dwUnused为一个保留位，未被使用。

UsbInstallDriver函数主要用于创建一个驱动程序加载所需的注册表信息。其函数原型为：

```
BOOL USBInstallDriver(LPCWSTR szDriverLibFile );
```

里面的唯一参数szDriverLibFile表示的就是dll驱动程序文件的名称。当函数成功执行的时候，该函数返回True，否则返回False。

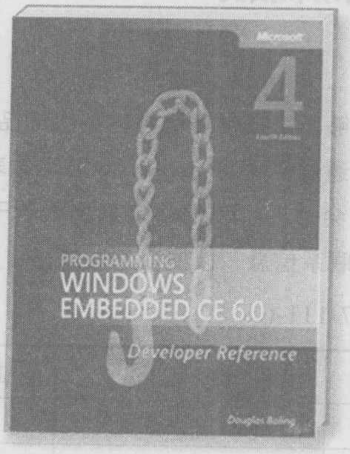
USBUninstallDriver函数主要用于释放驱动程序所占用的资源，以及删除USBInstallDriver函数创建的注册表等。其函数原型为：

```
BOOL USBUnInstallDriver(void);
```

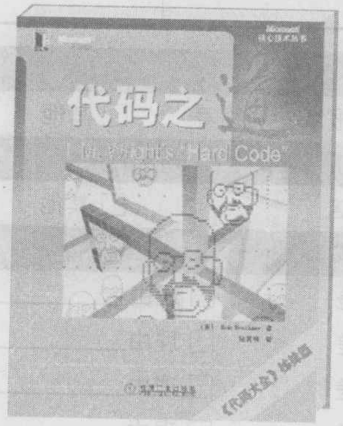
同样，当函数成功执行的时候，函数返回True，否则返回False。

在Windows CE中，USBD通过查询固定的注册表项来加载驱动程序。这个配置在注册表项HKEY_LOCAL_MACHINE\Drivers\USB\LoadClients下面，每个驱动程序的子键格式为Group1_ID\Group2_ID\Group3_ID\DriverName，如果注册表信息与USB设备信息符合，USBD就会加载此驱动程序。

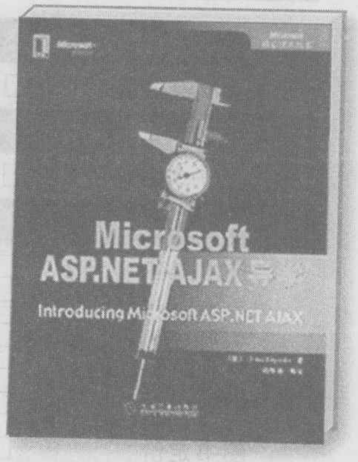
微软经典图书



Windows CE 6.0 开发者参考
作者：Douglas Boling
书号：978-7-111-26040-0
出版时间：2009年
微软官方权威图书！人手必备的经典之作！



代码之道
作者：Eric Brechner
译者：陆其明
书号：978-7-111-25167-5



Microsoft ASP.NET AJAX 导学
作者：Dino Esposito 译者：成保栋
书号：978-111-24372-4
定价：39.00元

Windows 核心编程
作者：Jeffrey Richter 译者：黄陇、李虎
书号：978-7-111-23791-4
定价：99.00元

Silverlight 2 导学
作者：Laurence Moroney
书号：978-7-111-25404-1
定价：49.00元

专业成就人生
立体服务大众

HZ BOOKS

www.hzbook.com

填写读者调查表　加入华章书友会
获赠精彩技术书　参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名：　　　　　　　　　　　　　　书号：7-111-(　　　　　　　　)

姓名：	性别：☐男　☐女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

☐朋友推荐　☐书店　☐图书目录　☐杂志、报纸、网络等　☐其他

2. 您从哪里购买本书：

☐新华书店　☐计算机专业书店　☐网上书店　☐其他

3. 您对本书的评价是：

技术内容	☐很好	☐一般	☐较差	☐理由____
文字质量	☐很好	☐一般	☐较差	☐理由____
版式封面	☐很好	☐一般	☐较差	☐理由____
印装质量	☐很好	☐一般	☐较差	☐理由____
图书定价	☐太高	☐合适	☐较低	☐理由____

4. 您希望我们的图书在哪些方面进行改进？

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

6. 您有没有写作或翻译技术图书的想法？

☐是，我的计划是________　☐否

7. 您希望获取图书信息的形式：

☐邮件　☐信函　☐短信　☐其他____

请寄：北京市西城区百万庄南街1号　机械工业出版社　华章公司　计算机图书策划部收
邮编：100037　电话：(010) 88379512　传真：(010) 68311602　E-mail: hzjsj@hzbook.com

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):
{
  "filename": "V0lORE9XUyBFTUJFRERFRCBDRSA2LjDnqIvluo/orr7orqHlrp7miJhfMTIyNDgzMjQuemlw",
  "filename_decoded": "WINDOWS EMBEDDED CE 6.0\u7a0b\u5e8f\u8bbe\u8ba1\u5b9e\u6218_12248324.zip",
  "filesize": 126182853,
  "md5": "49ed2aa19c69aeea6bef1be83f228594",
  "header_md5": "e3bbeb167d0c01f49aceba6d0ce6ad62",
  "sha1": "43259a450be53d5ae53e41358d1d30171164d617",
  "sha256": "7b1af6a1764034da9b63c7cb67acc56684c41b04cb9889f079245441be2bb276",
  "crc32": 2636502996,
  "zip_password": "",
  "uncompressed_size": 131825923,
  "pdg_dir_name": "WINDOWS EMBEDDED CE 6.0\u2502\u2560\u2568\u2265\u2554\u03a6\u255d\u255e\u2569\u2561\u2552\u255c_12248324",
  "pdg_main_pages_found": 328,
  "pdg_main_pages_max": 328,
  "total_pages": 345,
  "total_pixels": 2061488160,
  "pdf_generation_missing_pages": false
}